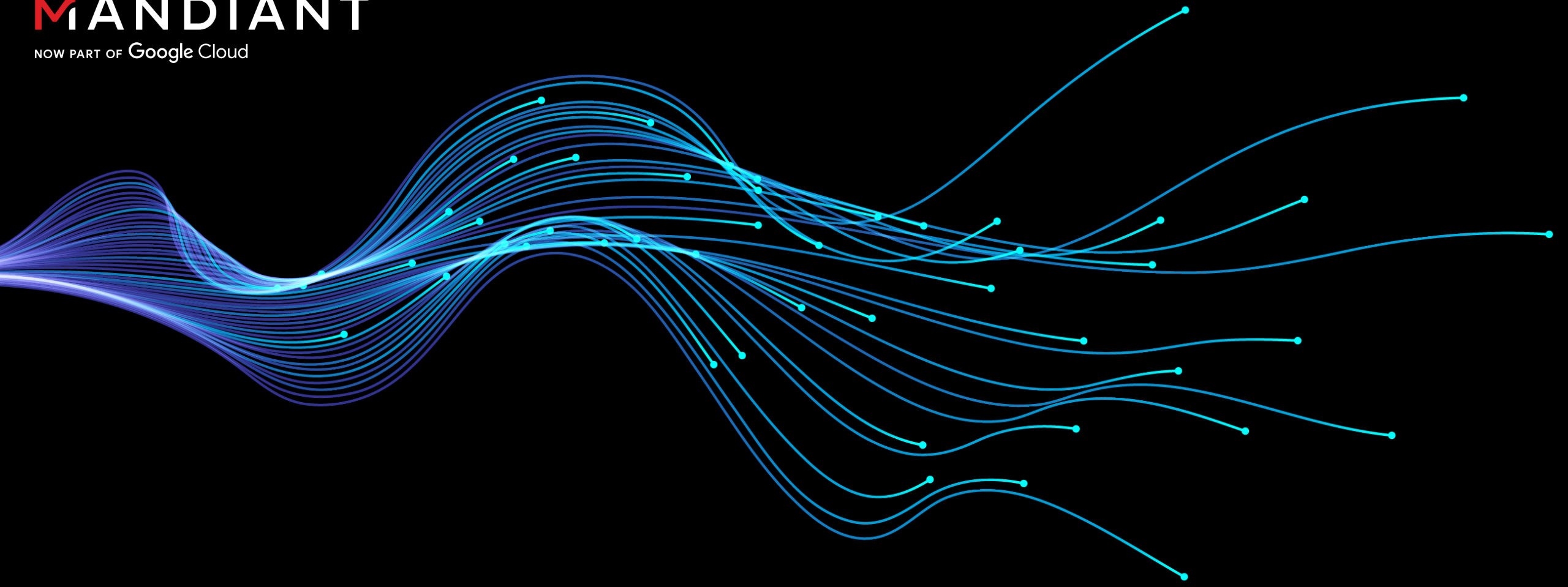
MANDIANT
NOW PART OF Google Cloud



# Fuzzing at Mach Speed

*Uncovering MacOS IPC Vulnerabilities with Dillon Franke*

# Who Am I?

MANDIANT
NOW PART OF Google Cloud

**CURRENTLY**

**Senior Proactive Security Consultant**
(Pentesting)
- *Application Security*
- *Source Code Reviews*
- *Embedded Device Assessments*

**PREVIOUSLY**

**FLARE Offensive Task Force (OTF)**
(Reverse Engineering)
- *Malware reversing*
- *Searching for exploits used in the wild*
- *0-day vulnerability research*
- *Exploit development*

**STUDIED**

**Bachelor's & Master's in Computer Science at Stanford University**
*Security and Systems Engineering*

**HOBBIES**

**Playing Guitar**
**Cycling in the San Francisco Bay Area**
**Hacking (obviously)**

# Overview

*Join me as a I dive into my process searching for low-level vulnerabilities in MacOS over the past year.*

Crash Course on Fuzzing and IPC Mechanisms

The Attack Cycle

Next Steps

Q&A

CRASH COURSE

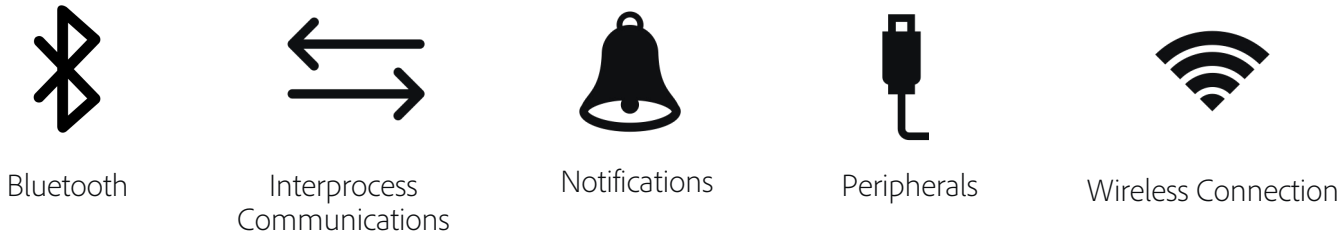
# What is Fuzzing?

Fuzzing is sending unexpected **inputs** to a **system** in the hopes of making something unexpected happen

CRASH COURSE

# What is an Attack Vector?

An attack vector is a channel to send an **input** to a **system**



CRASH COURSE

# What is an Attack Vector?

An attack vector is a channel to send an **input** to a **system**

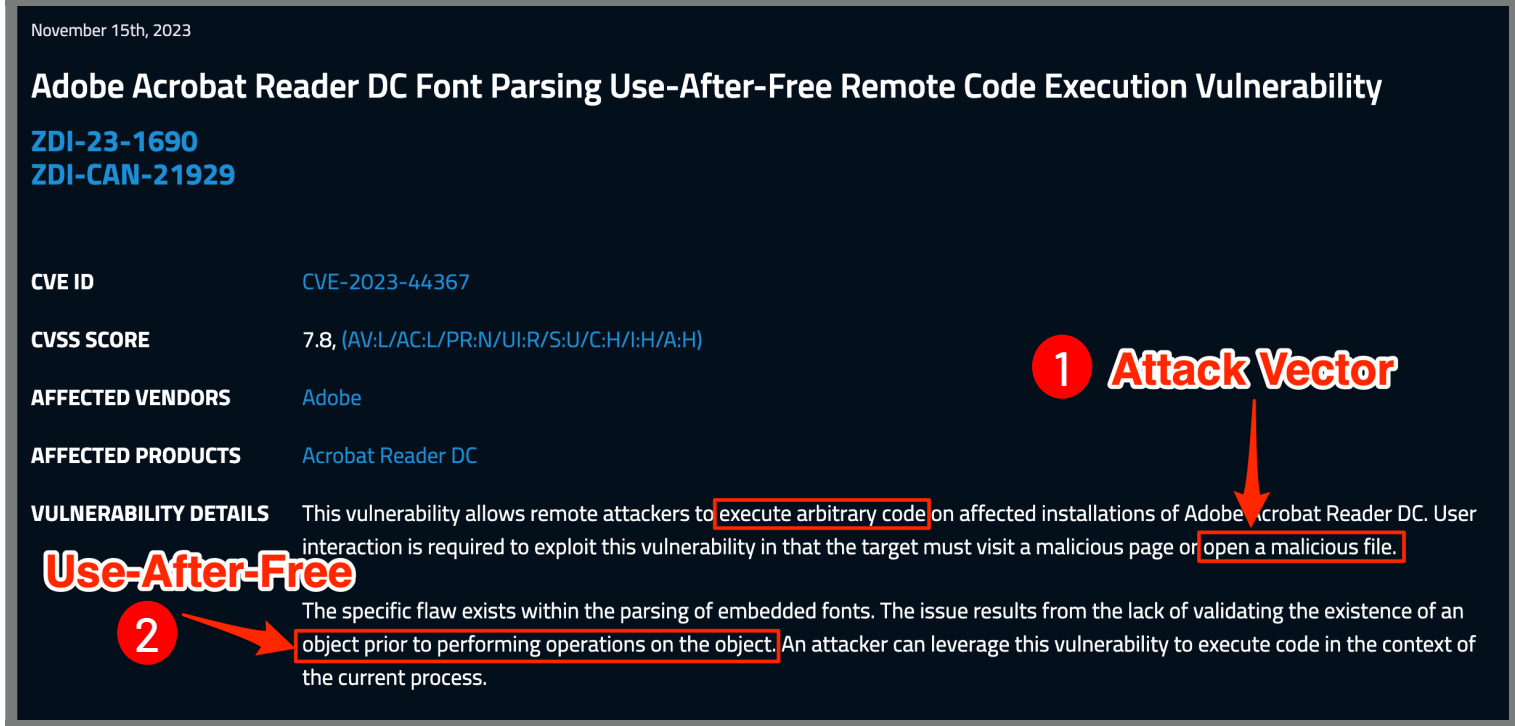
| | |
|---|---|
| Adobe Acrobat | Open PDF Functionality |
| Google Search | Query Parameter (https://google.com?query=**<INPUT>**) |
| Smart Watch | Bluetooth Data Handling |

CRASH COURSE

# Why Fuzz?

1. In memory-unsafe languages, (C/C++) we want to send input that causes a crash

2. Depending on the type of crash, our input might be able to trigger:
   - Buffer Overflow
   - Heap Overflow
   - Use-After-Free
   - Double Free
   - Memory Leak (bypass ASLR)



CRASH COURSE

# Different Types of Fuzzing

**Mutation-Based Fuzzing:** Modify existing inputs to create new ones, then send them to the program

**Grammar-Based Fuzzing:** Generate inputs based on specified rules defining the structure of valid inputs

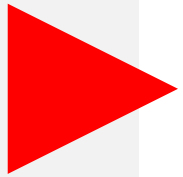
CRASH COURSE

# What is the XNU Kernel?

XNU (X is Not Unix) is the kernel that powers macOS.

CRASH COURSE

# What is the XNU Kernel?



XNU (X is Not Unix) is the kernel that powers macOS.

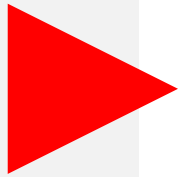
**Mach Layer:** Responsible for low-level tasks like thread management, interprocess communication (IPC), and memory management.

**BSD Layer:** Handles higher-level POSIX tasks, like file system, network, and security.

**I/O Kit:** A framework for developing device drivers, designed with a model resembling object-oriented programming.

CRASH COURSE

# What is the XNU Kernel?



XNU (X is Not Unix) is the kernel that powers macOS.

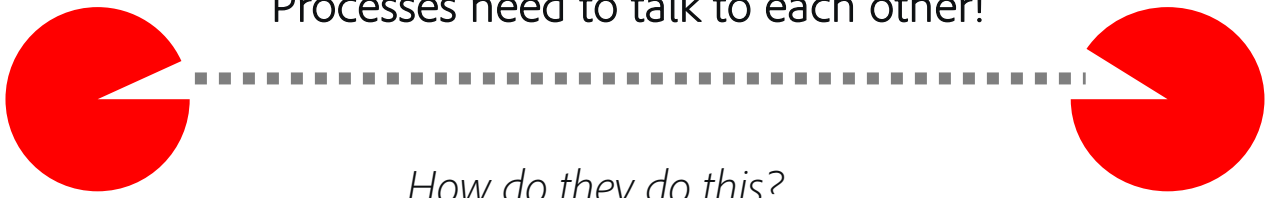
**Mach Layer:** Responsible for low-level tasks like thread management, **interprocess communication (IPC)**, and memory management.

**BSD Layer:** Handles higher-level POSIX tasks, like file system, network, and security.

**I/O Kit:** A framework for developing device drivers, designed with a model resembling object-oriented programming.

CRASH COURSE

# What are Interprocess Communications?

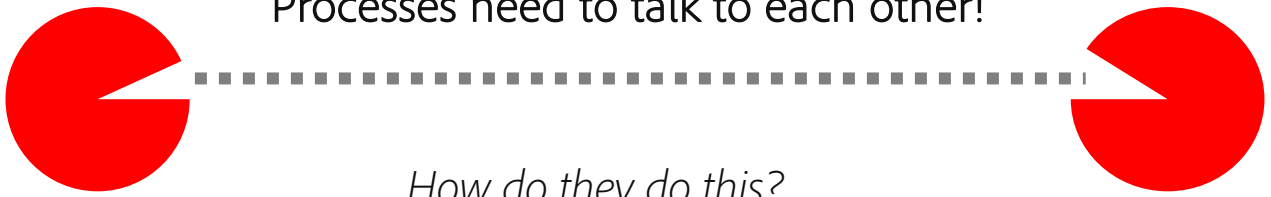



*How do they do this?*

Mach Messages
Mach exceptions
Unix signals
Unnamed pipes
Named pipes (fifos)
XSI/System V IPC
POSIX IPC
Distributed Objects
Apple Events
Core Foundation IPC mechanisms

CRASH COURSE

# What are Interprocess Communications?




*How do they do this?*

**Mach Messages**
Mach exceptions
Unix signals
Unnamed pipes
Named pipes (fifos)
XSI/System V IPC
POSIX IPC
Distributed Objects
Apple Events
Core Foundation IPC mechanisms

Lowest level IPC mechanism and the direct basis for many higher level mechanisms

CRASH COURSE

# What are Mach Ports?

## An IPC message queue, managed by the kernel

**Port Right:** Handle to a port that allows sending or receiving messages to the port

**Receive Right:** Allows receiving a mach port's messages

**Send Right:** Allows sending messages to a mach port

**Send Once:** Allows sending a single message to a mach port

CRASH COURSE

# What are Mach Ports?

> An IPC message queue, managed by the kernel

**Port Right:** Handle to a port that allows sending or receiving messages to the port

**Receive Right:** Allows receiving a mach port's messages

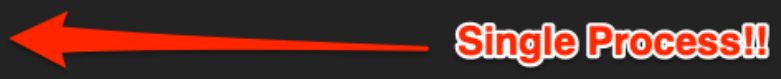
**Send Right:** Allows sending messages to a mach port

**Send Once:** Allows sending a single message to a mach port

```
❯ lsmp -h
Usage: lsmp -p <pid> [-a|-v|-h]
Lists information about mach ports.
Please see man page for description
of each column.
```

```
Process (135) : kextd
  name      ipc-object    rights     flags   boost  reqs  recv  send sonce oref  qlimit  msgcount  context
---------  ----------  ----------  --------  -----  ----  ----- ----- ----- ----  ------  --------  ----------
0x00000103  0xdce4a79b  send        --------          ---             2
0x00000203  0xdd0c45e3  recv        --------      0   ---       1             N       5         0  0x00000000
0x00000307  0xd6247d5b  send        --------          ---              54
0x00000403  0xdd0c41f3  recv        --------      0   ---       1             N       5         0  0x00000000
0x00000503  0xdd0c564b  recv        --------      0   ---       1             N       5         0  0x00000000
0x00000603  0xdce4a8eb  send        --------          ---              1
0x00000703  0xdd0c56f3  recv        --------      0   ---       1             N       5         0  0x00000000
0x00000803  0xd624781b  send        --------          ---              1
0x00000903  0xdcc335a3  recv,send   ---GS---      0   ---       1     2       Y       5         0  0x00000000
0x00000a03  0xdcc690e3  recv,send   ---GS---      0   ---       1     1       Y       5         1  0x00000000
                  +     send        --------          ---              1       <-
0x00000b03  0xdcc6957b  send        --------          ---              1       ->      1         0  0x00000000
0x00000c03  0xdcc69623  send        --------          ---              1       ->      1         0  0x00000000
0x00000d0f  0xde2da7db  recv        --------      0   ---       1             Y       5         0  0x00000000
                  +     send        --------          D--              1       <-
0x00000e07  0xd6248fbb  send        --------          ---              1       ->      32        0  0x00000000
0x00000f03  0xdcaeff13  send        --------          ---              1       ->      6         0  0x00000000
0x00001003  0xdcaefbcb  send        --------          ---              1
0x00001103  0xd6247e03  send        --------          ---              1
0x00001203  0xdcc6abcb  recv,send   --------      0   ---       1     1       Y       5         0  0x00000000
0x00001303  0xd779214b  send        --------          ---              6       ->      128       0  0x00000000
0x00001403  0xdd0c2cf3  send        --------          ---              1
0x00001507  0xdcb5718b  send        --------          ---              1       ->      6         0  0x00000000
```

```
total     = 845
SEND      = 841
RECEIVE   = 5
SEND_ONCE = 0
PORT_SET  = 0
DEAD_NAME = 0
DNREQUEST = 0
VOUCHERS  = 0
```



CRASH COURSE

# Establishing a Mach Connection

**Bootstrap Server**

- A mach port to help establish connections with other mach ports
- By default, all processes have a send right to the bootstrap server

**Mach Service**

- A mach port with a name that is registered with the Bootstrap Server (e.g. **com.apple.moose**)

**Communicating with a Service**

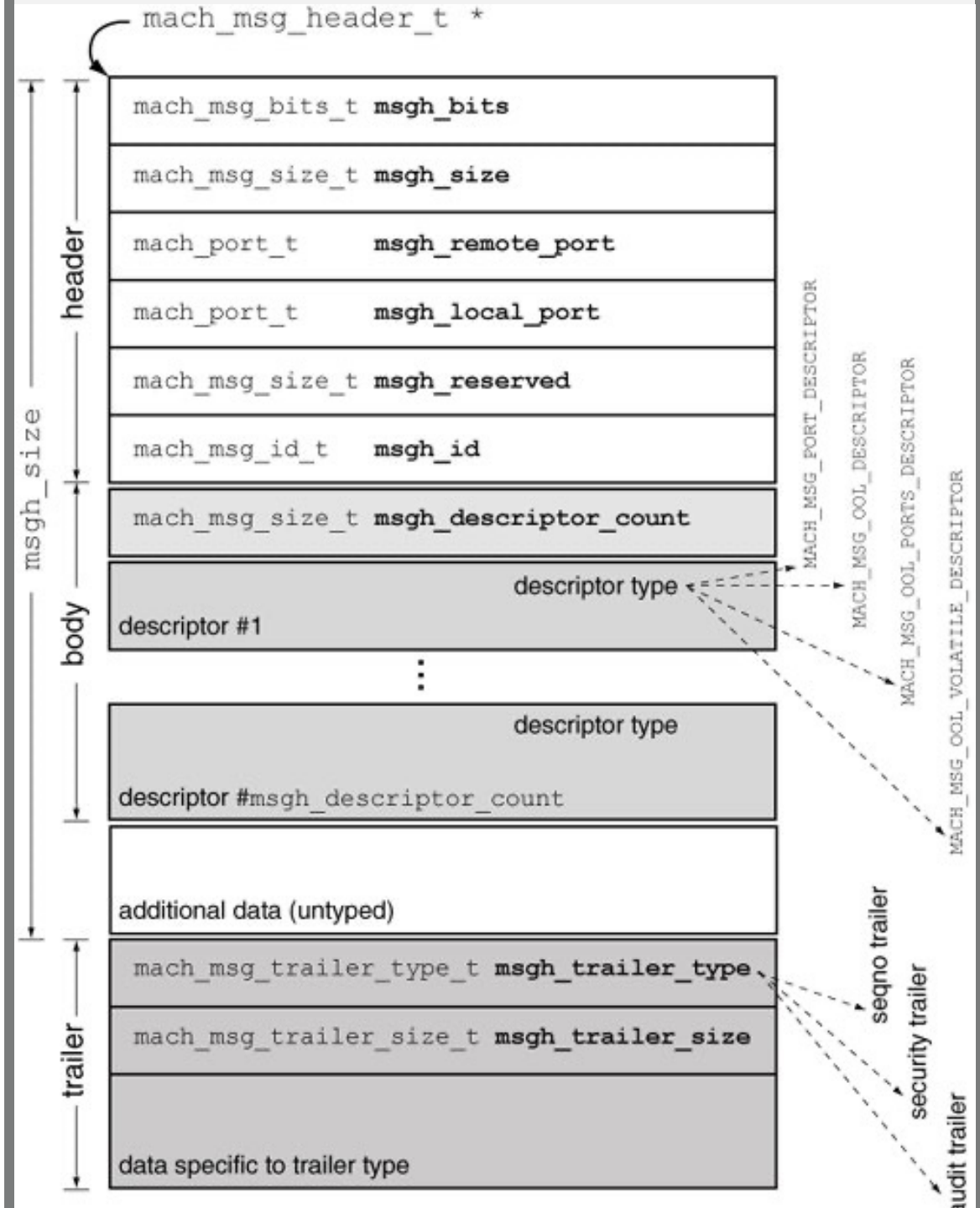
1. Alice allocates a new mach port with a receive right
2. Alice registers her service using a specific name **com.apple.moose**

   *By registering, Alice is giving the bootstrap server a send right to the port Alice has a receive right to*
3. Bob asks the bootstrap server for the service named **com.apple.moose** and the server gives Bob a copy of the send right for Alice's mach port
4. Bob can now send messages to Alice's mach port for Alice to receive

CRASH COURSE

# What are Mach Messages?

A struct used to exchange data between mach ports



CRASH COURSE

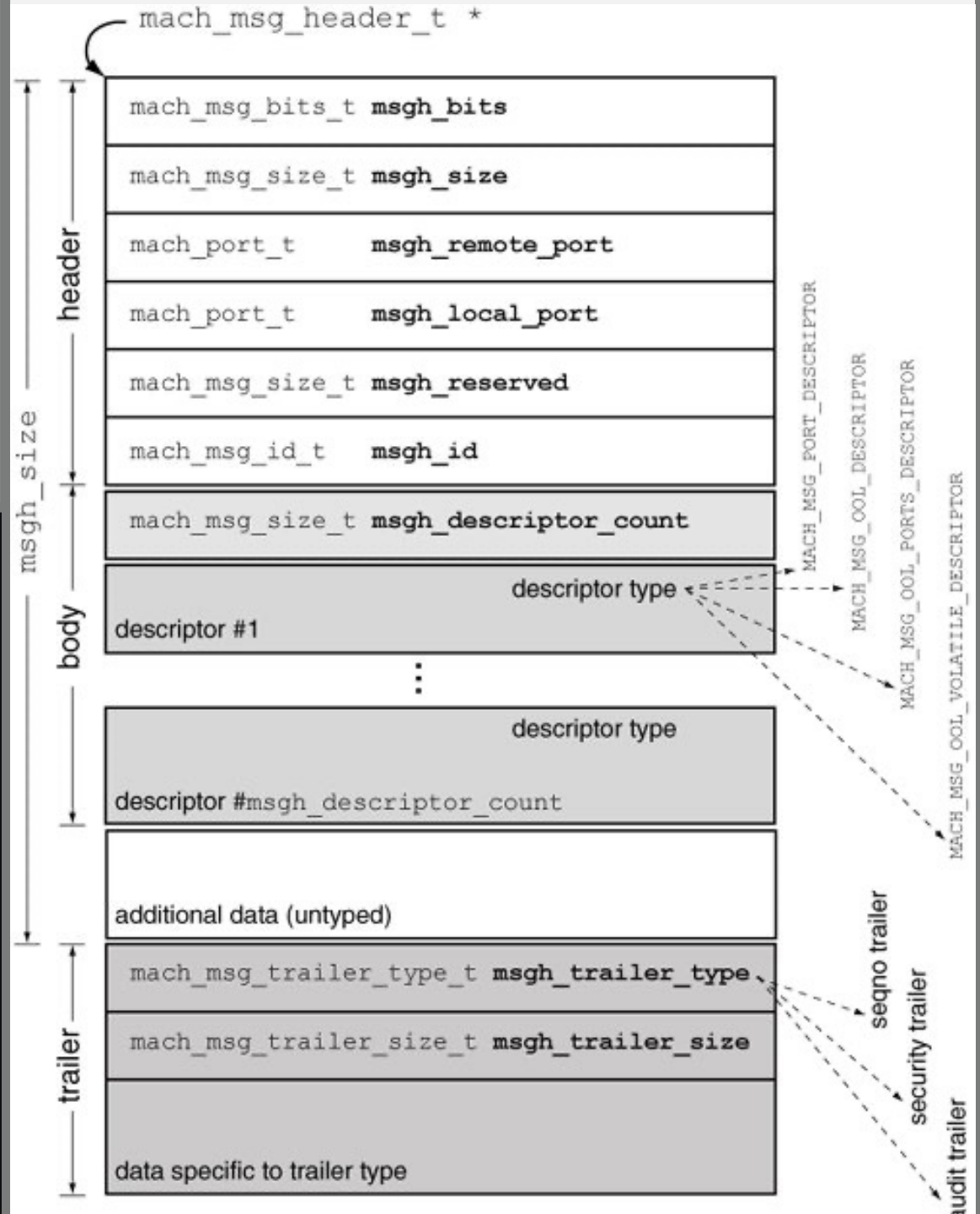
# What are Mach Messages?

A struct used to exchange data between mach ports

*Sending/Receiving Mach Messages*

```
/*
* Routine: mach_msg
* Purpose:
* Send and/or receive a message. If the message operation
* is interrupted, and the user did not request an indication
* of that fact, then restart the appropriate parts of the
* operation silently (trap version does not restart).
*/
__WATCHOS_PROHIBITED __TVOS_PROHIBITED
extern mach_msg_return_t mach_msg(
  mach_msg_header_t *msg,
  mach_msg_option_t option,
  mach_msg_size_t send_size,
  mach_msg_size_t rcv_size,
  mach_port_name_t rcv_name,
  mach_msg_timeout_t timeout,
  mach_port_name_t notify
);
```

Option specifies send/receive!



THE ATTACK CYCLE

# The (Memory Corruption) Attack Cycle

Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

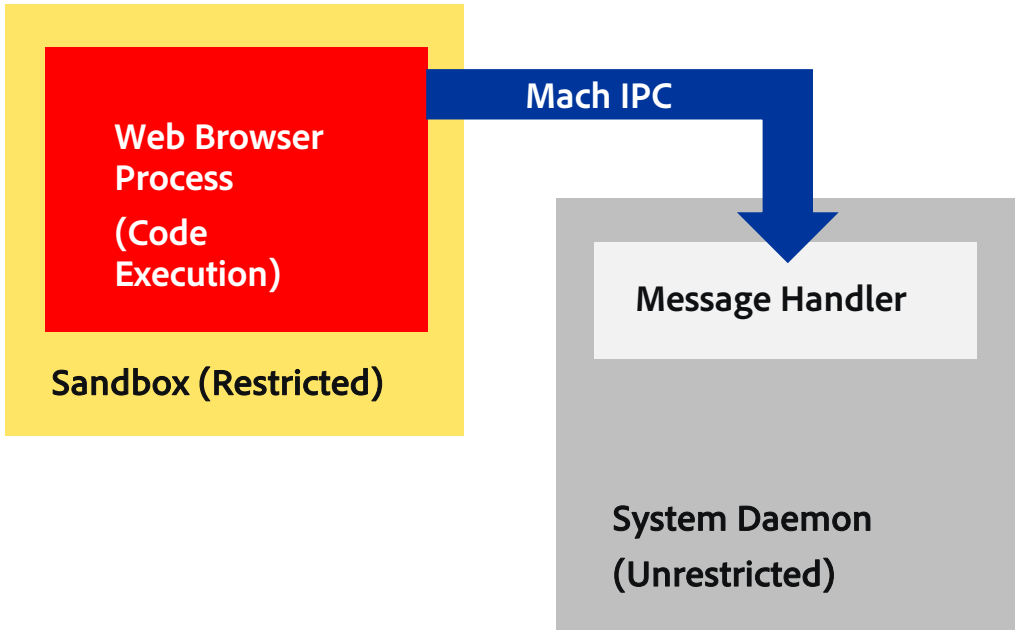
Identify Relevant Crashes

THE ATTACK CYCLE

# Abusing Mach Messages

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

## Sandbox Escape

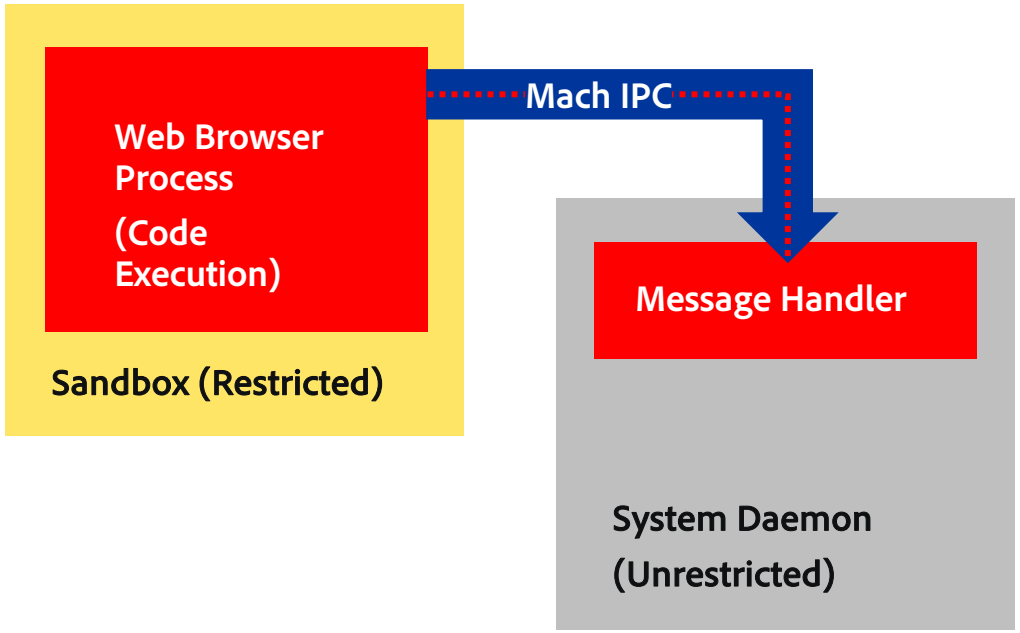


THE ATTACK CYCLE

# Abusing Mach Messages

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

## Sandbox Escape

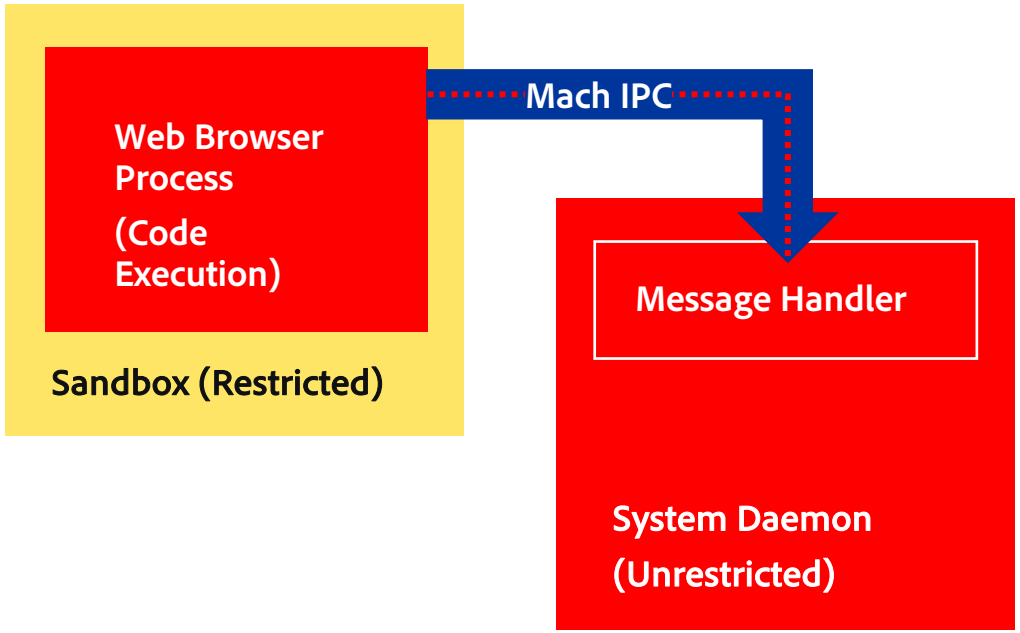


THE ATTACK CYCLE

# Abusing Mach Messages

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

## Sandbox Escape



Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

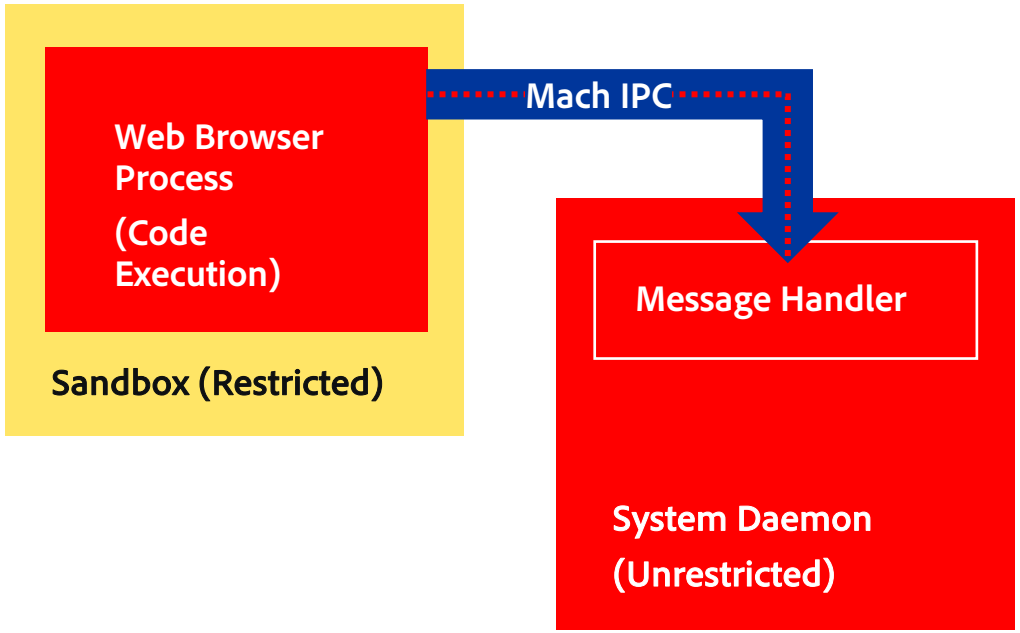
Fuzz and Produce Crashes

Identify Relevant Crashes

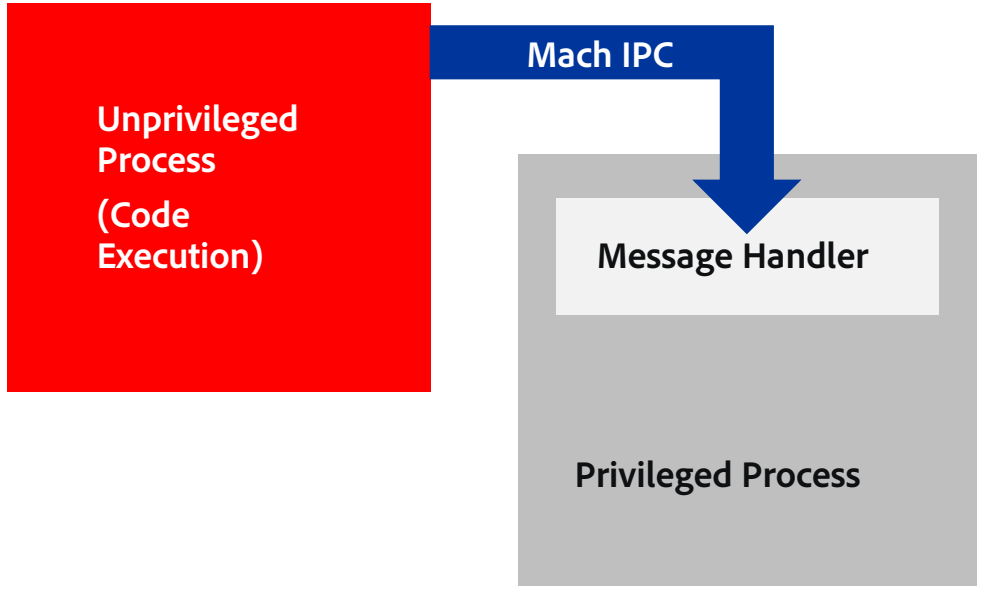
THE ATTACK CYCLE

# Abusing Mach Messages

## Sandbox Escape




## Privilege Escalation

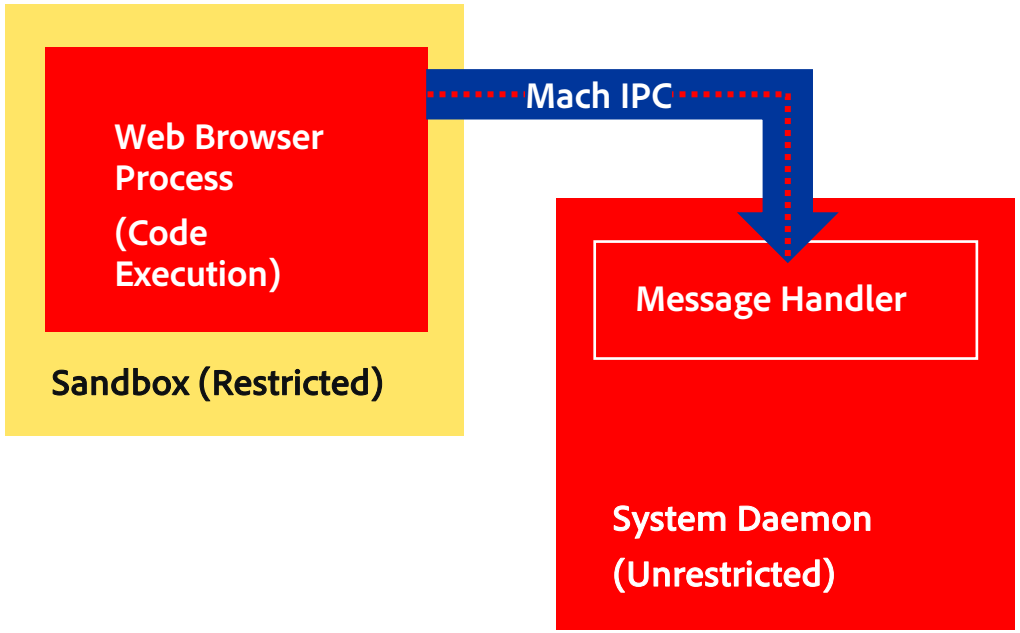




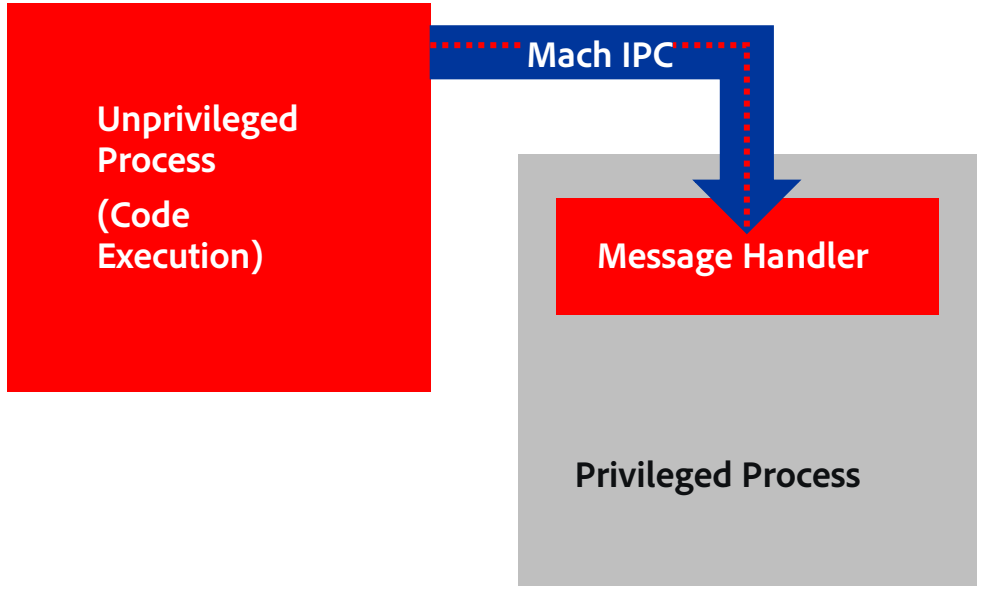
# Abusing Mach Messages



## Sandbox Escape




## Privilege Escalation

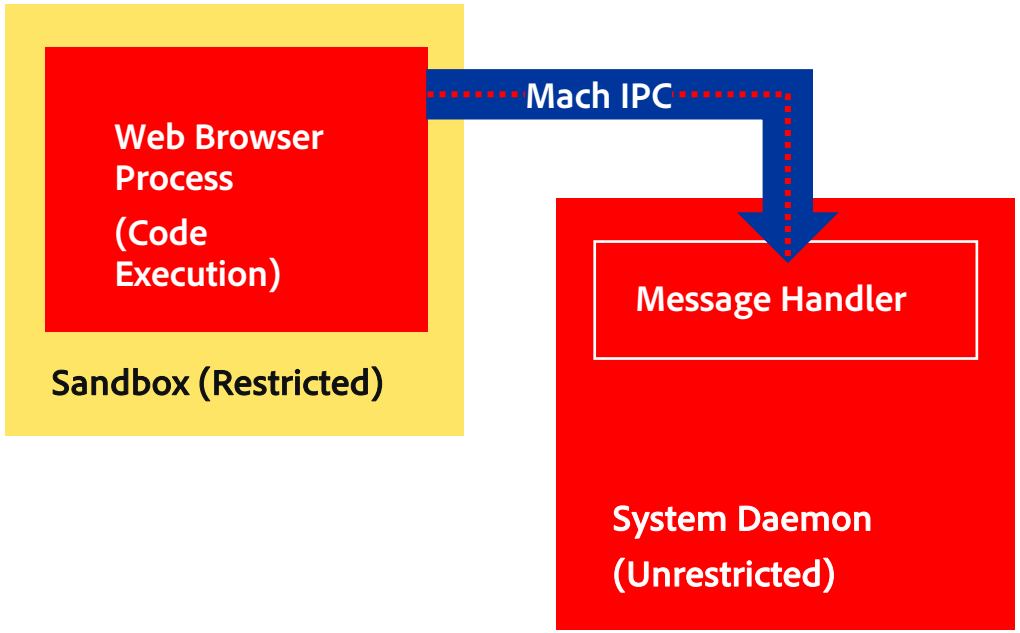




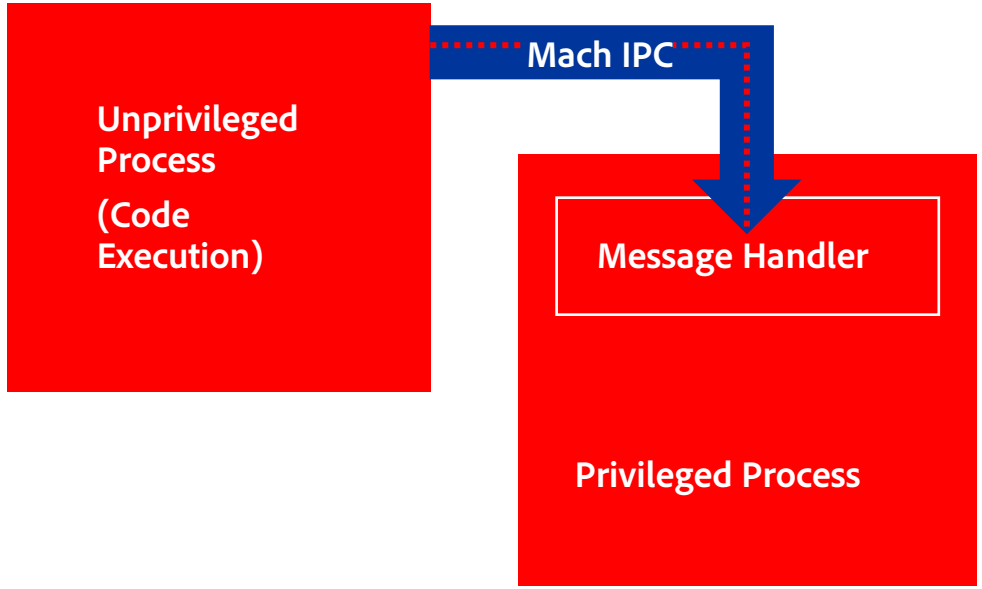
# Abusing Mach Messages



## Sandbox Escape




## Privilege Escalation





# Finding Sandbox-Allowed Communications

*How do we know what processes could allow an escape?*



**sbtool**: https://newosxbook.com/src.jl?tree=listings&file=/sbtool.c

- Use built-in **sandbox_check()** function to determine which mach services a process can send to
- Message handlers we can send to → potential for sandbox escapes

```
❯ ./sbtool 2813 mach
com.apple.logd
com.apple.xpc.smd
com.apple.remoted
com.apple.metadata.mds
com.apple.coreduetd
com.apple.apsd
com.apple.coreservices.launchservicesd
com.apple.bsd.dirhelper
com.apple.logind
com.apple.revision
…Truncated…
```



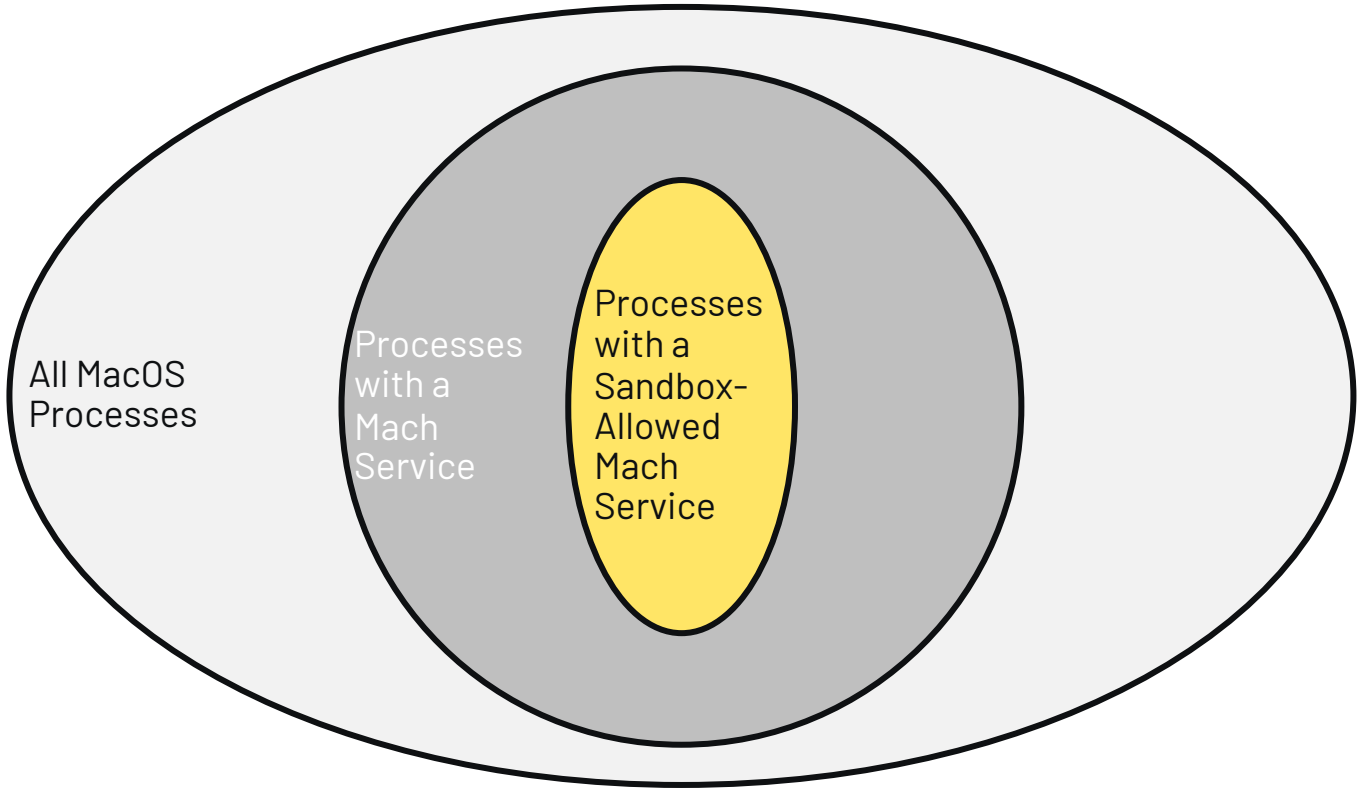
# Finding Sandbox-Allowed Communications

*How do we know what processes could allow an escape?*



**sbtool**: https://newosxbook.com/src.jl?tree=listings&file=/sbtool.c

- Use built-in **sandbox_check()** function to determine which mach services a process can send to
- Message handlers we can send to → potential for sandbox escapes





# Previous Mach Research



*BlackHat: Breaking the Chrome Sandbox with Mojo*

- https://i.blackhat.com/USA-22/Wednesday/US-22-Roettger-Breaking-the-Chrome-Sandbox-with-Mojo.pdf
- Race condition + DoS == RCE

*A Methodical Approach to Browser Exploitation*

- http://blog.ret2.io/2018/06/05/pwn2own-2018-exploit-development/
- Safari sandbox escape via mach IPC messages == RCE

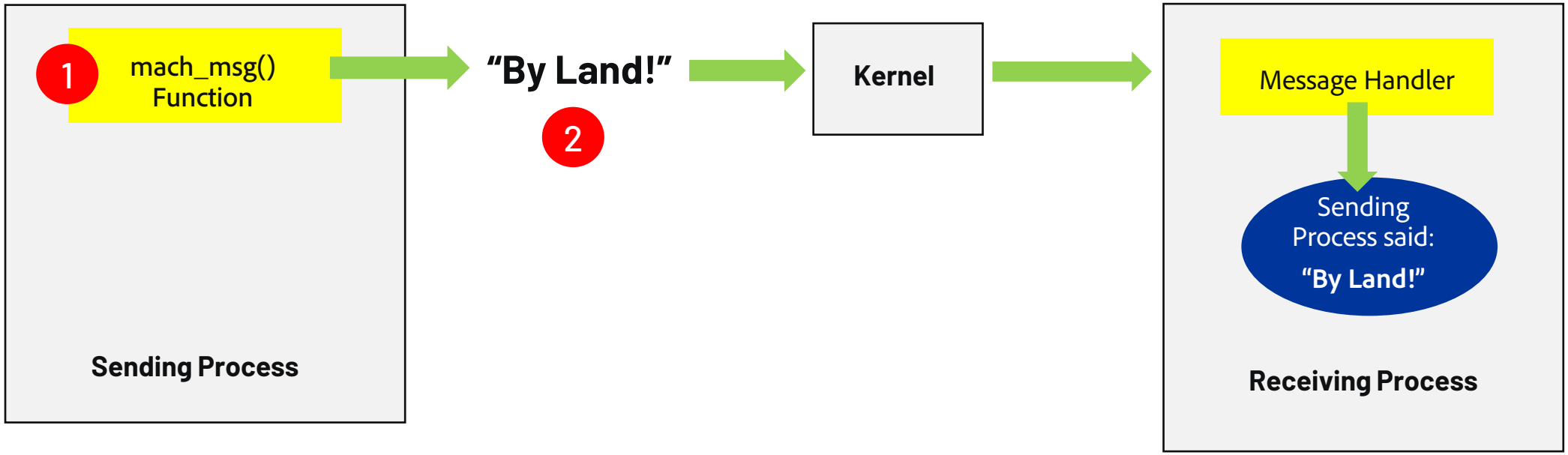


# Finding an Entry Point



1. We know that **mach_msg()** is used to send mach messages from one process to another

2. Why not just modify real mach messages being sent?



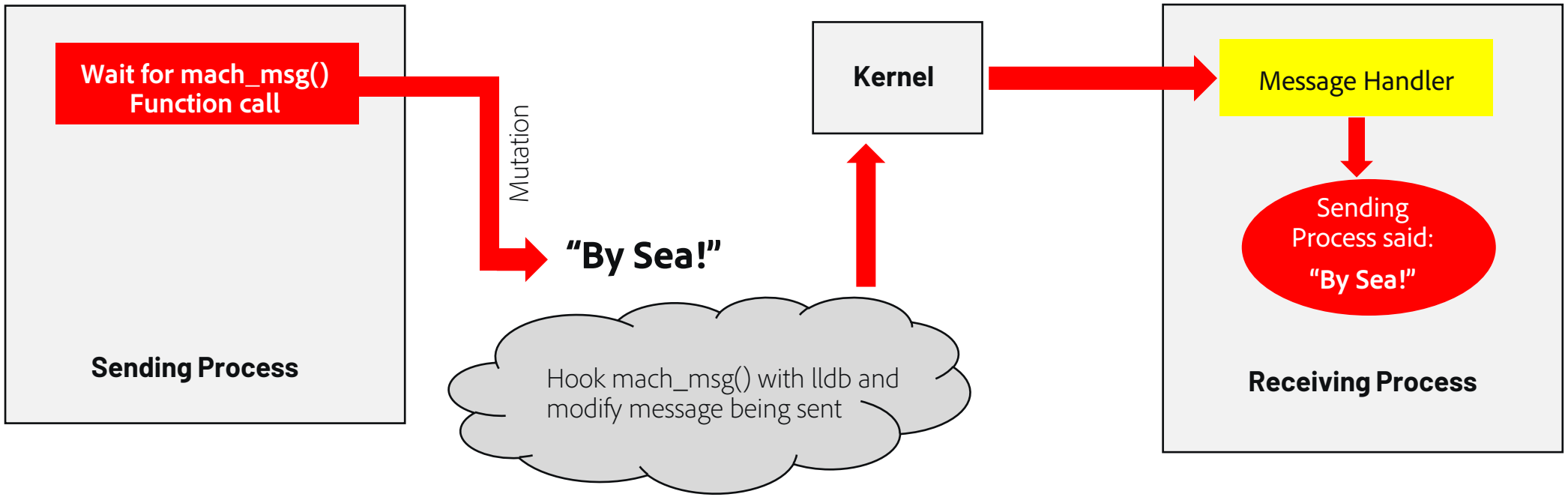


# Finding an Entry Point



1. We know that **mach_msg()** is used to send mach messages from one process to another

2. Why not just modify real mach messages being sent?





THE ATTACK CYCLE

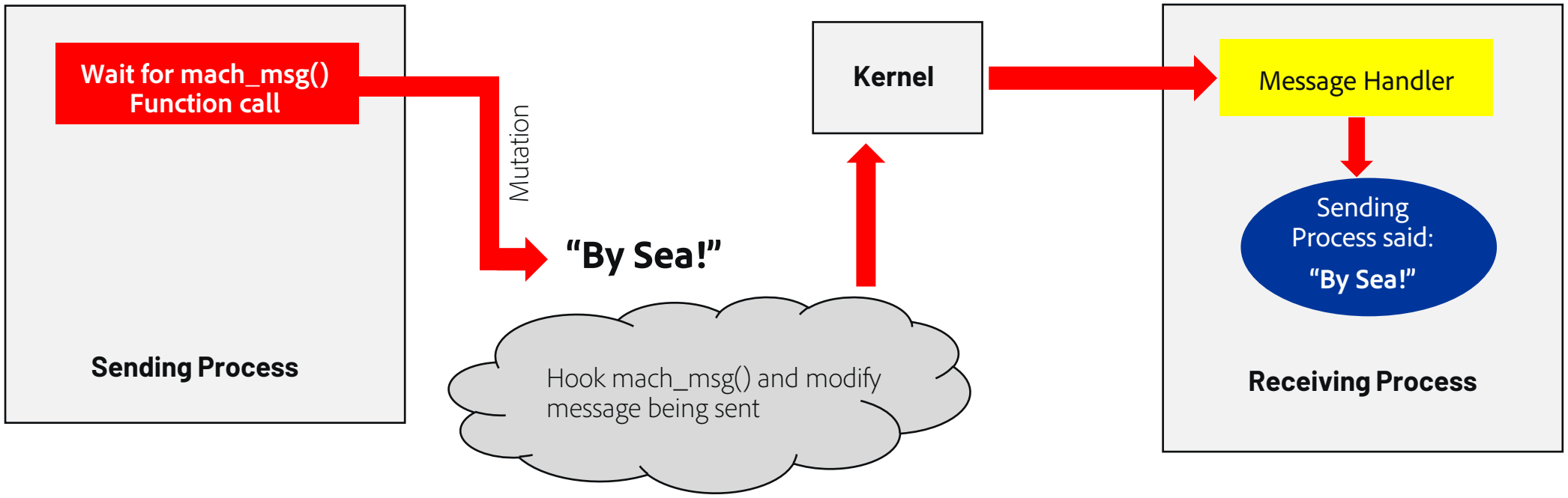
# Finding an Entry Point

**Pros:**

- Simple
- Similar to end exploit

**Cons:**

- Slow *(At mercy of the application to send messages)*
- Many points of potential failure
- Two different process spaces (code coverage difficult)
- Difficult to determine which message caused crash

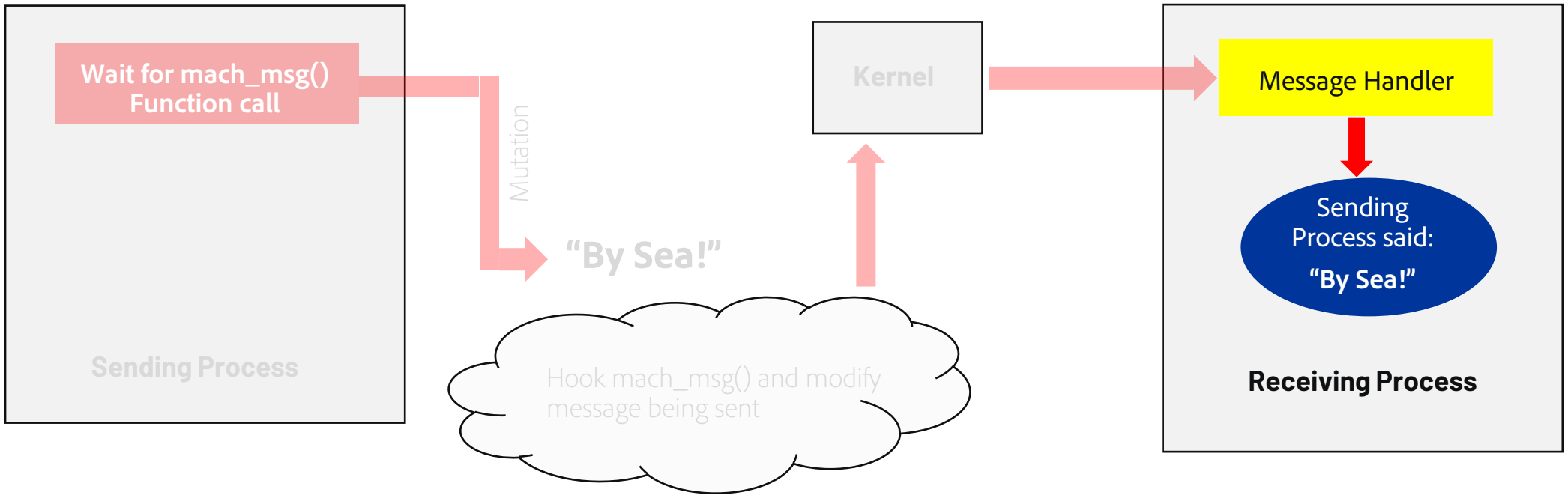




# Finding an Entry Point



Instead of waiting for **mach_msg()** to be called, what if we write a program to call it ourselves?

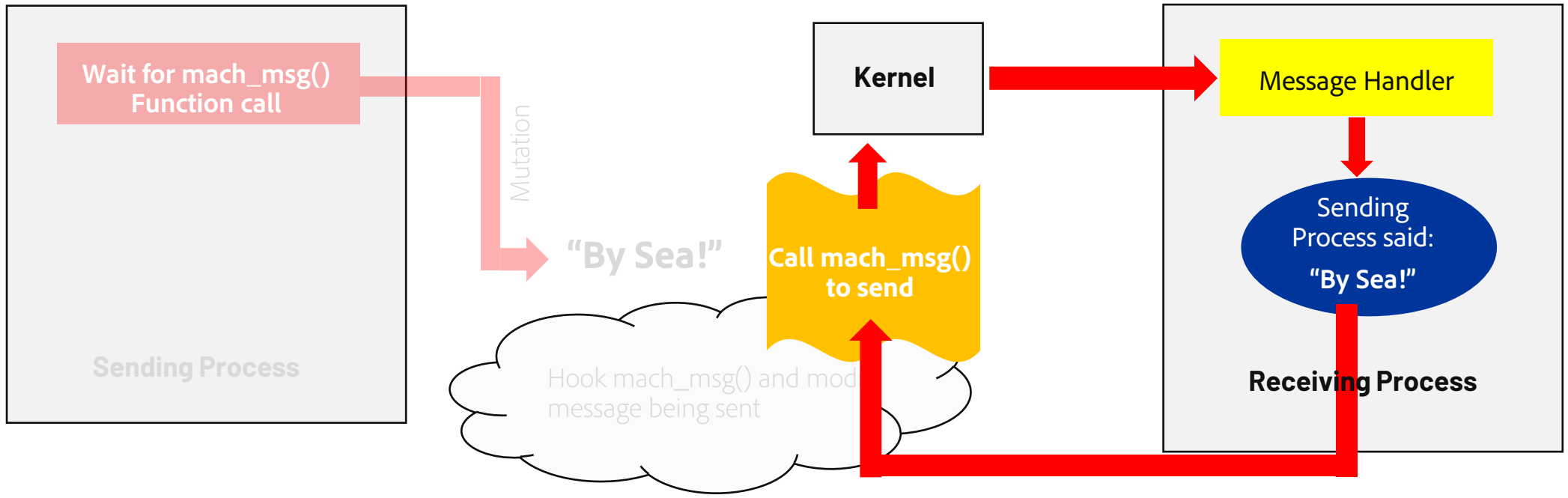




THE ATTACK CYCLE

# Finding an Entry Point

Instead of waiting for **mach_msg()** to be called, what if we write a program to call it ourselves?





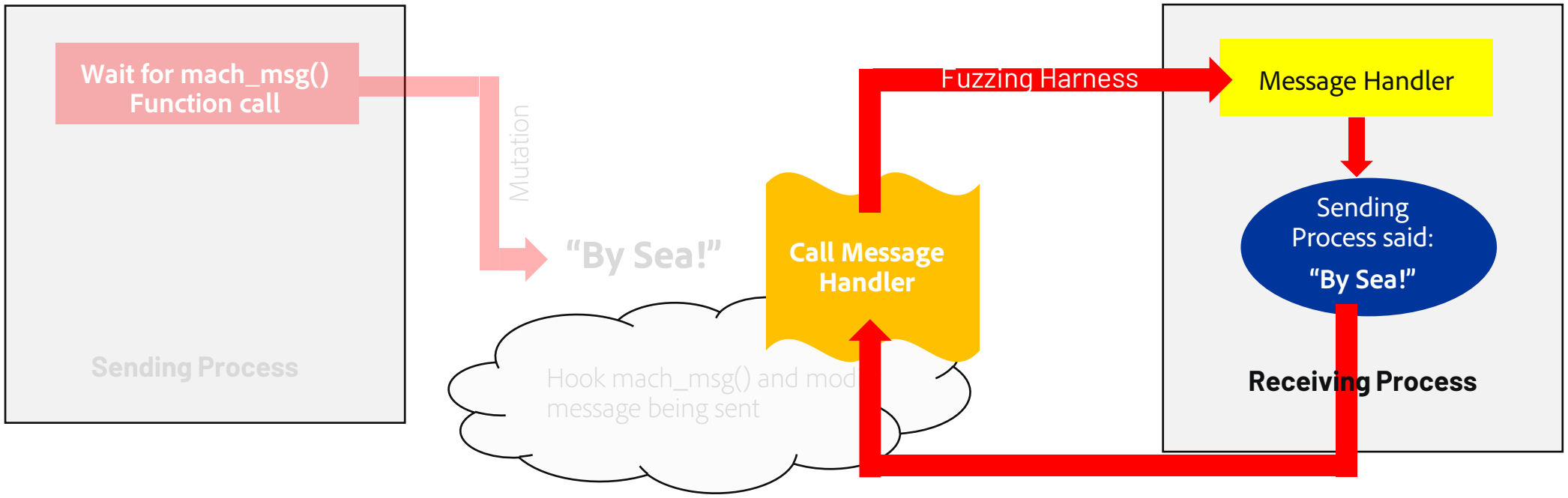
THE ATTACK CYCLE

# Finding an Entry Point

Instead of waiting for **mach_msg()** to be called, what if we write a program to call it ourselves?

Even Better: What if we just called the message handler directly?

**Getting "close" to the system of interest**





THE ATTACK CYCLE

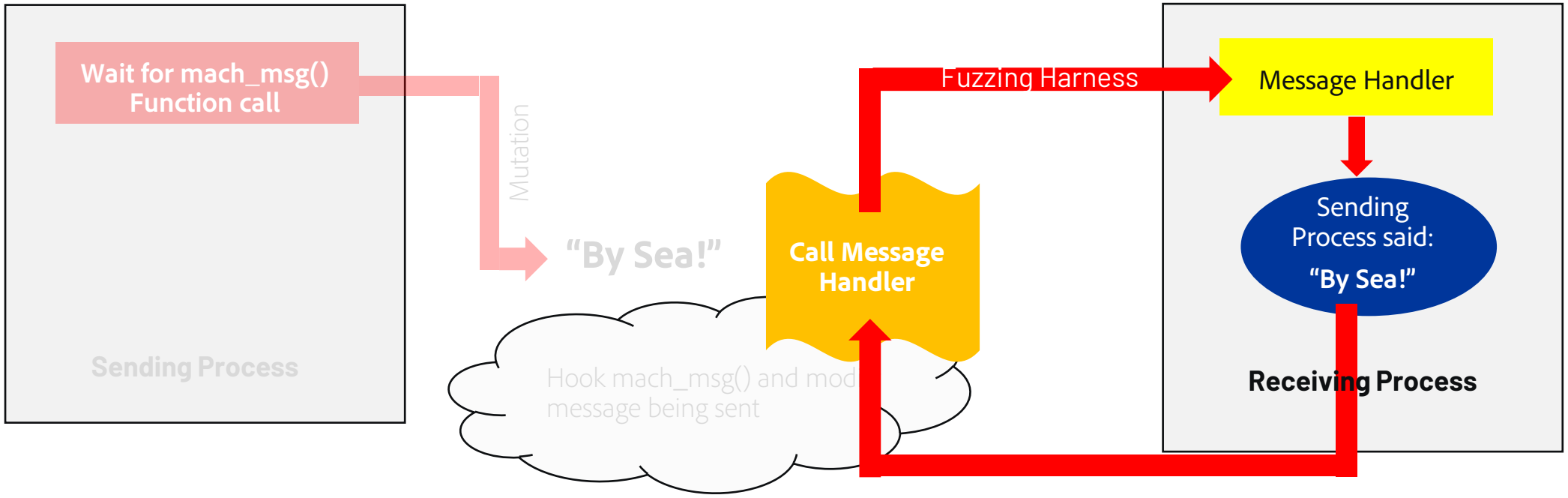
# Finding an Entry Point

**Pros:**

- Very fast
- Same process space easy for instrumentation/code coverage
- Easy to know which input caused crash/replicate

**Cons:**

- Different from end exploit
- Might have to invoke initialization routines



THE ATTACK CYCLE

# We have an attack vector – but what should we send?

Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

Identify Relevant Crashes

Sending totally random data is not likely to produce meaningful crashes

- Exception handlers
- Input validation

**We need to identify examples of valid mach messages (e.g. “corpus building”)**

THE ATTACK CYCLE

# Prep-Work

Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

Identify Relevant Crashes

## A number of things to take into consideration when we start debugging on MacOS

1. Setting up a MacOS virtual machine
2. Disabling System Integrity Protection (SIP)
   - `csrutil disable`
3. Disabling ReportCrash
4. Disabling Sleep
   - `systemsetup -setsleep Never`
5. Much more information provided: Jeremy Brown - Summer of Fuzz: MacOS - DEF CON 29 AppSec Village

THE ATTACK CYCLE

# Finding the Mach Message Handler

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

## 1 Find a mach service of interest

In our case, will be services sandboxed processes can communicate with

Let's focus on **com.apple.audio.coreaudiod**

- Handles all interactions with audio hardware
- Privileged process
- Allowed to send mach messages from many processes

THE ATTACK CYCLE

# Finding the Mach Message Handler

Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

Identify Relevant Crashes

## 2 Find the binary that implements the mach service

- **com.apple.audio.coreaudiod** registered with **launchd**
- Spawns **/usr/sbin/coreaudiod**
- Mach server handled by CoreAudio Framework

```
(lldb) image list
[  0] D5BCB621-948E-308C-AF2C-88489D5569FA 0x000000010f332000 /usr/sbin/coreaudiod
[  1] BB7A0970-8C62-3DCE-A7A2-5CEC9C501F11 0x00007ff80894f000 /usr/lib/dyld
[  2] 66BBA3CA-BCE1-32F8-8269-99FAC92469FC 0x00007ff8123d6000 /System/Library/PrivateFrameworks/caulk.framework/Versions/A/caulk
[  3] 97A3CD09-7112-376C-9613-7F38D4CF8C41 0x00007ff80ac99000 /System/Library/Frameworks/CoreAudio.framework/Versions/A/CoreAudio
[  4] BEB5FC0B-7196-3C1D-A59A-F62ADA98F592 0x00007ff808ce4000 /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
```

```
fuzzychicken@Fuzzys-Mac HALB_MIGServer_server % stat /System/Library/Frameworks/CoreAudio.framework/Versions/A/CoreAudio
stat: cannot stat '/System/Library/Frameworks/CoreAudio.framework/Versions/A/CoreAudio': No such file or directory
```

THE ATTACK CYCLE

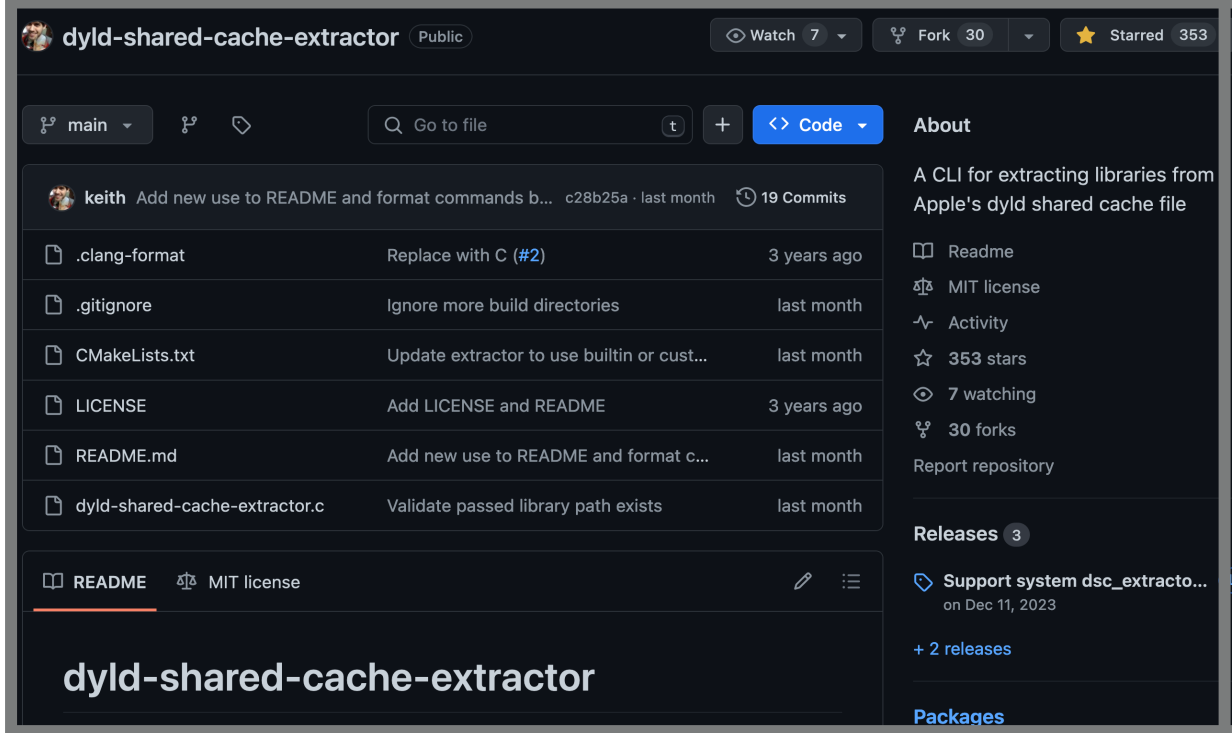
# Finding the Mach Message Handler

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

## 3 Extract the binary from the dyld shared cache

- **Dyld shared cache:** Starting with Big Sur, most framework binaries are not on disk
- We can extract them!
- https://github.com/keith/dyld-shared-cache-extractor

```
build› ./dyld-shared-cache-extractor /System/Volumes/Preboot/C
ryptexes/OS/System/Library/dyld/dyld_shared_cache_x86_64h extr
acted-binaries
extracted 0/2505
extracted 1/2505
extracted 2/2505
extracted 3/2505
extracted 4/2505
extracted 5/2505
extracted 6/2505
extracted 7/2505
extracted 9/2505
extracted 8/2505
```



Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

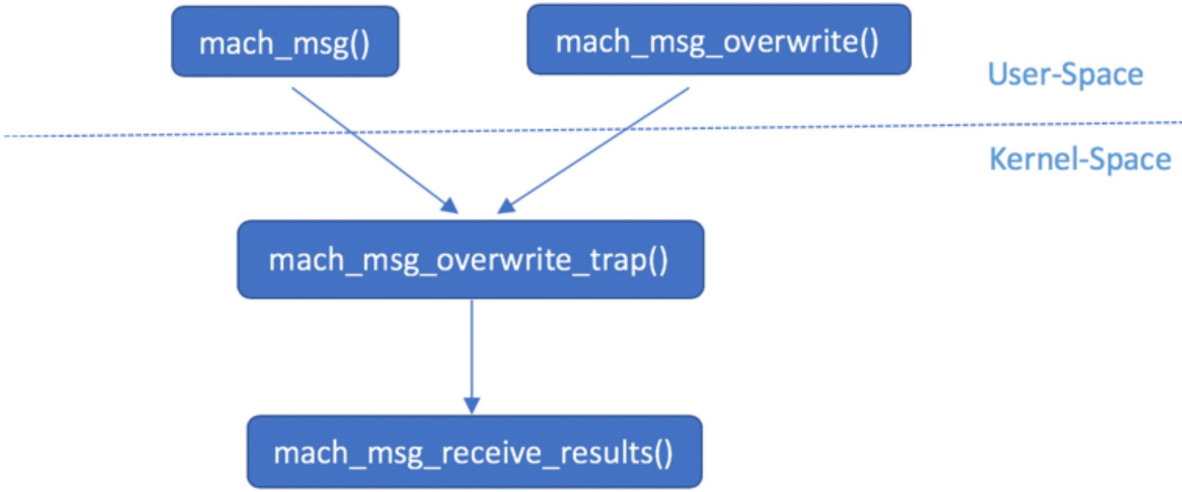
Identify Relevant Crashes

THE ATTACK CYCLE

# Finding the Mach Message Handler

## 4 Find function implementing mach receive functionality

- Wait, isn't this just **mach_msg()**?
  - Non-blocking, traps to kernel when a message is received
- Need to perform kernel debugging if we want to intercept incoming mach messages
  - This has been done: https://www.fortinet.com/blog/threat-research/inspect-mach-messages-in-macos-kernel-mode--part-ii--sniffing-th
- Kernel debugging cons:
  - We see all mach messages, difficult to isolate target process
  - Two-machine debugging required
- Is there an easier way?



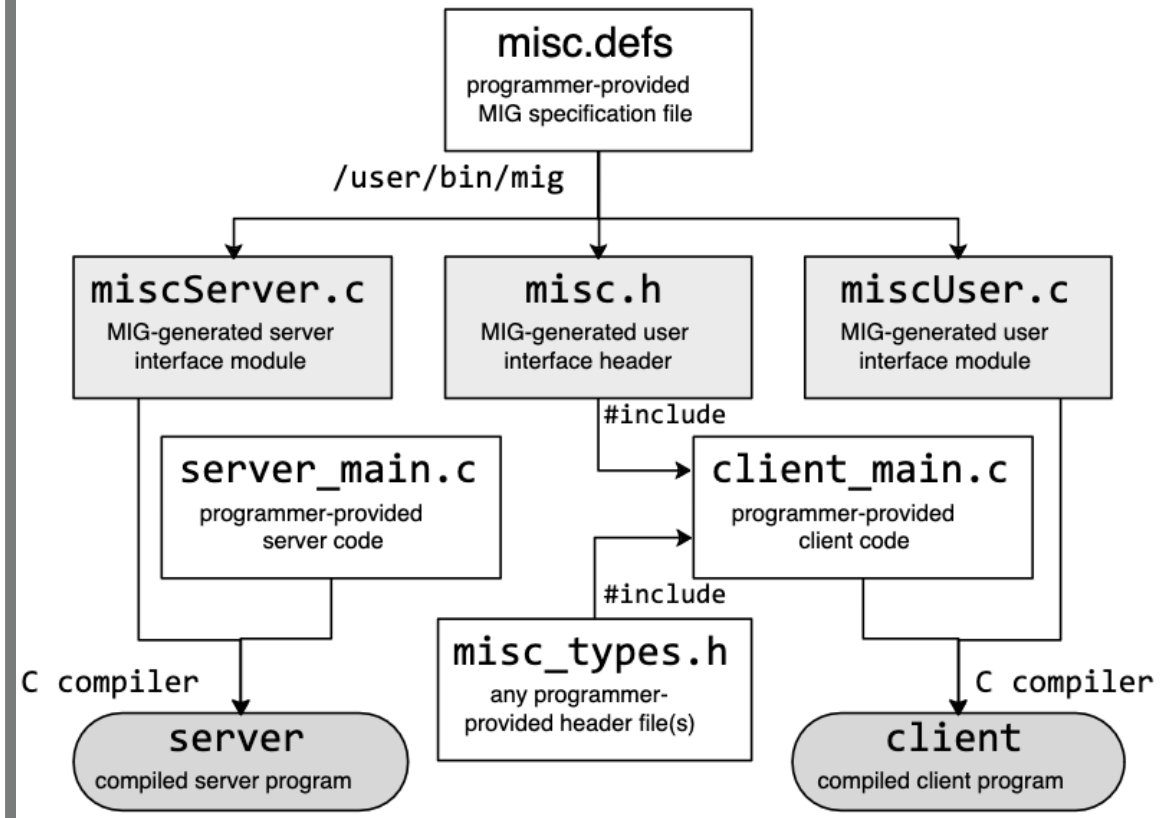
THE ATTACK CYCLE

# Finding the Mach Message Handler

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

## Mach Interface Generator (MIG)

- Apple provides MIG to more easily write RPC handlers and clients
- Interface Definition Language (IDL) compiler
- Abstracts much of the mach IPC layer away
- What if we searched for MIG-generated routines and dumped their incoming mach messages?




https://wcventure.github.io/FuzzingPaper/Paper/SRDS19_MachFuzzer.pdf

THE ATTACK CYCLE

# Finding the Mach Message Handler

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

## 3 Find function implementing mach receive functionality

- Hopper script: https://github.com/knightsc/hopper/blob/master/scripts/MIG%20Detect.py

```
build/framework-binaries> nm -m ./System/Library/Frameworks/CoreAudio.framework/Vers
ions/A/CoreAudio | grep -i subsystem

                 (undefined) external _CACentralStateDumpRegisterSubsystem (from Aud
ioToolboxCore)
00007ff8401adec0 (__DATA_CONST,__const) non-external _HALC_HALB_MIGClient_subsystem
00007ff8401adfd0 (__DATA_CONST,__const) non-external _HALS_HALB_MIGServer_subsystem
```

Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

Identify Relevant Crashes

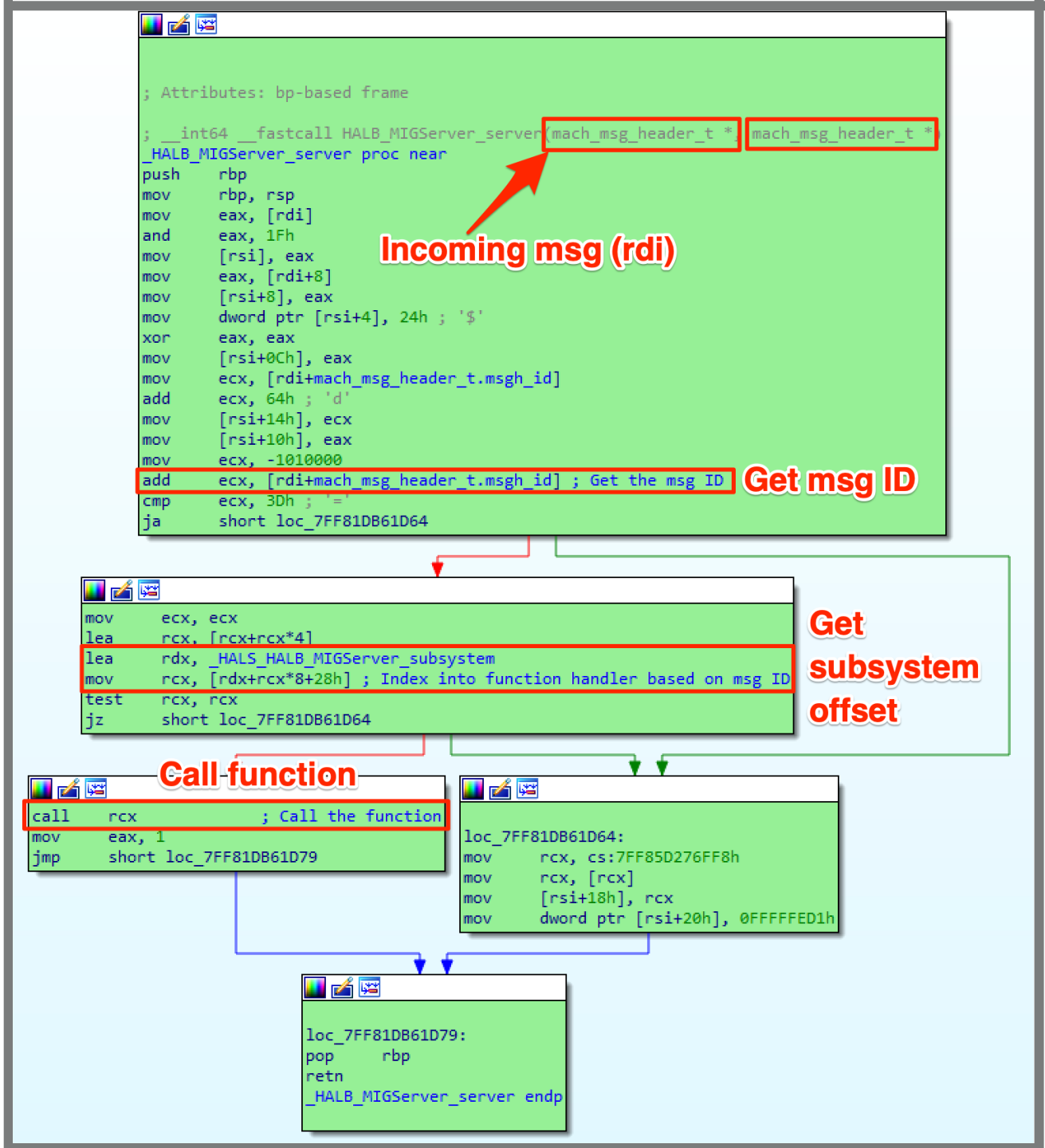
THE ATTACK CYCLE

# Finding the Mach Message Handler

## 3 Find function implementing mach receive functionality

_HALS_HALB_MIGServer_subsystem

- Function lookup table



Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

Identify Relevant Crashes

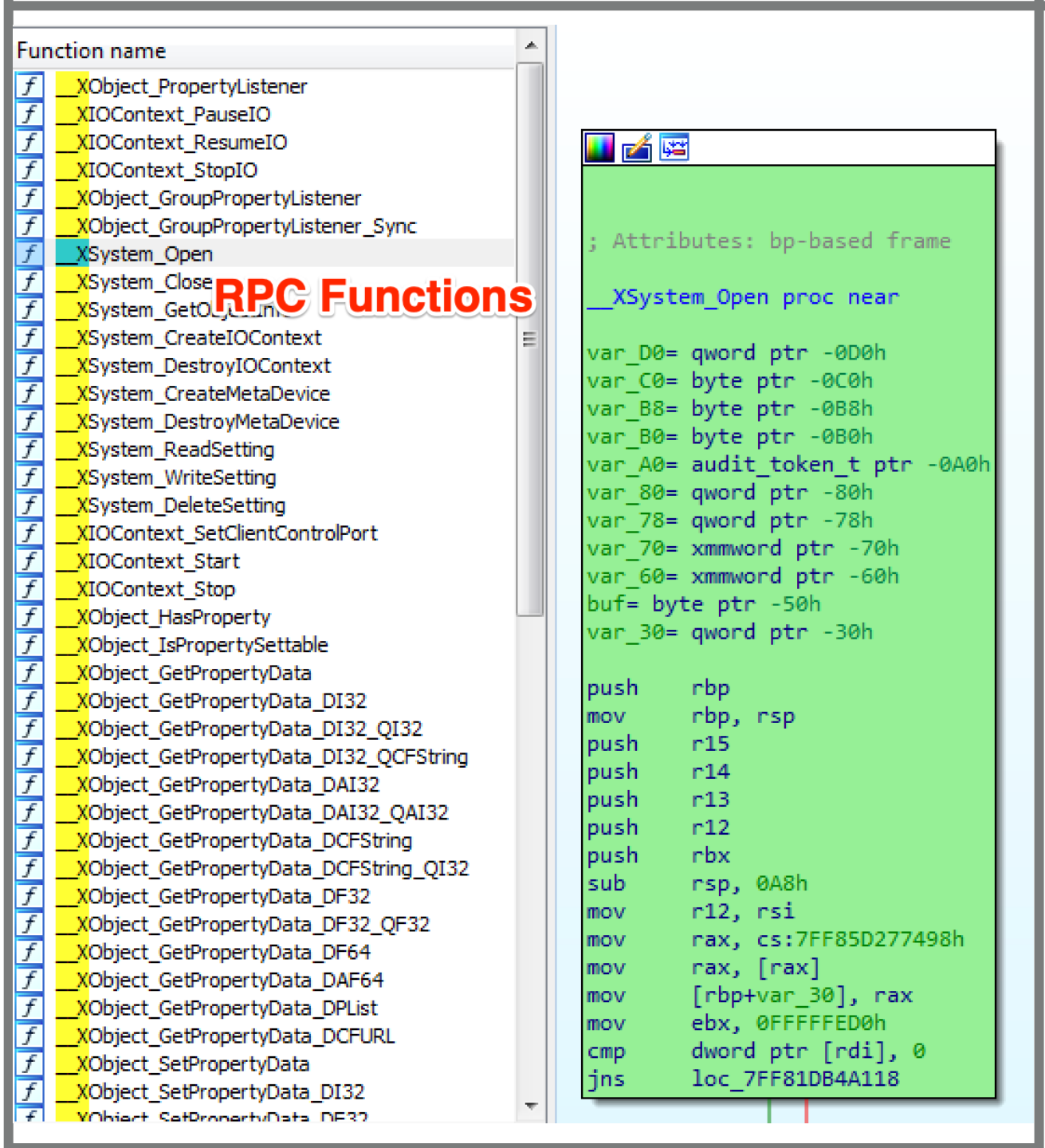
THE ATTACK CYCLE

# Finding the Mach Message Handler

## 3 Find function implementing mach receive functionality

_HALS_HALB_MIGServer_subsystem

- Function lookup table



THE ATTACK CYCLE

# Generate a Corpus of Inputs

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

**I wrote a simple script to hook onto the message handler using LLDB**

```
fuzzychicken@Fuzzys-Mac mach-fuzzing % sudo python3 subsystem_mach_msg_dumper.py -h
INFO Adding the LLDB Python library to PATH...
usage: subsystem_mach_msg_dumper.py [-h] -p PID -m MODULE -f FUNCTION

Attach to a process and dump a mach message passed to a specified function. The mach
 message should be passed as the first argument.

options:
  -h, --help            show this help message and exit
  -p PID, --pid PID     Process ID to attach to.
  -m MODULE, --module MODULE
                        Module loaded by the process.
  -f FUNCTION, --function FUNCTION
                        Function to set a breakpoint on.
```

THE ATTACK CYCLE

# Generate a Corpus of Inputs

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

```
fuzzychicken@Fuzzys-Mac mach-fuzzing %
```

THE ATTACK CYCLE

# What is a Fuzzing Harness?

- Identify an attack vector
- Generate a Corpus of Inputs
- **Create a Fuzzing Harness**
- Fuzz and Produce Crashes
- Identify Relevant Crashes

A **fuzzing harness** is code that allows you to send input through an attack vector.
(Call a desired function)



# Calling the Target Function



**Target Function: `_HALB_MIGServer_server`**

- Simple on Windows:
  - `HMODULE hModule = LoadLibrary(“libexample.dll”)`
  - `pFunction = GetProcAddress(hModule, “DesiredFunction”)`
- On MacOS, similar:
  - `void *lib_handle = dlopen(“libexample.dylib”, RTLD_LAZY)`
  - `pFunction = dlsym(lib_handle, “DesiredFunction”)`
- What if the symbol isn’t exported?
- Write your own Mach-O symbol parser
  - A talk for another time ☺



# Calling the Target Function



**Targe**

- Sim
  - HM
  - pF "D
- On M
  - vo RT
  - pF
- Wha
- Writ
  - A t

```
fuzzychicken@Fuzzys-Mac Release % ./Harness -l /System/Library/Frameworks/CoreAudio.framewo
rk/Versions/A/CoreAudio -s _HALB_MIGServer_server -f ~/mach-fuzzing/subsystem_messages/Core
Audio/HALB_MIGServer_server/a37747c4812a6baf1e4f5e[illegible]3d78d4c3

/System/Library/Frameworks/CoreAudio.framework/Versions/A/CoreAudio loaded at 0x1bbcd000

FUZZ Mach Msg:------ MACH MSG HEADER ------
msg_bits: 4370
msg_size: 48
msg_remote_port: 106187
msg_local_port: 67075
msg_voucher_port: 0
msg_id: 1010013
------ MACH MSG BODY (24 bytes) ------
0x0 0x0 0x0 0x0 0x1 0x0 0x0 0x0 0x66 0x0 0x0 0x0 0x76 0x73 0x63 0x6c 0x62 0x6f 0x6c 0x67 0x
0 0x0 0x0 0x0

Calling the function...
Result: 1

RETURNED Mach Msg:------ MACH MSG HEADER ------
msg_bits: 18
msg_size: 36
msg_remote_port: 106187
msg_local_port: 0
msg_voucher_port: 0
msg_id: 1010113
------ MACH MSG BODY (12 bytes) ------
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
fuzzychicken@Fuzzys-Mac Release %
```

Function (Attack Vector)

Mach Message (Input)

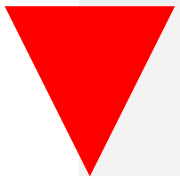
Return Mach Message

Calendar

THE ATTACK CYCLE

# What is a Fuzzer?

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- **Fuzz and Produce Crashes**
- Identify Relevant Crashes

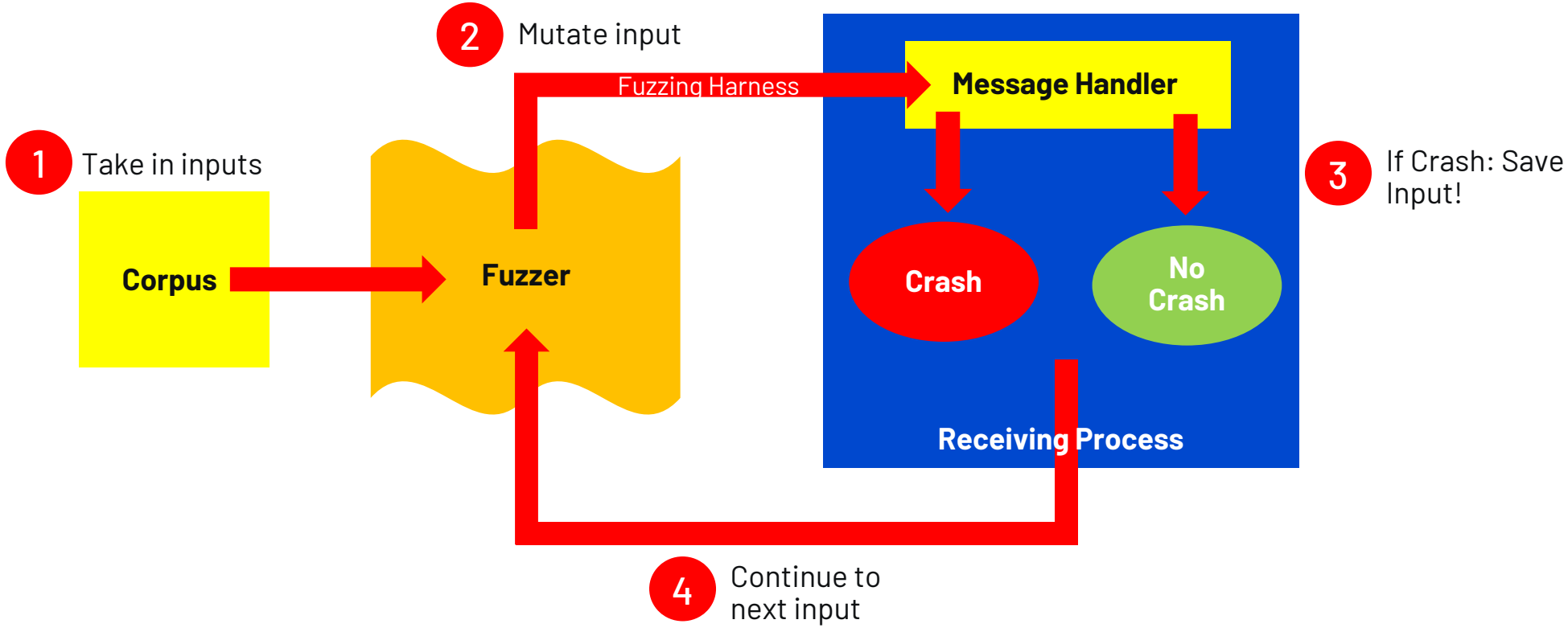


A **fuzzer** is a program that generates inputs to be sent to a system and monitors for crashes.



THE ATTACK CYCLE

# What is a Fuzzer?



THE ATTACK CYCLE

# The Need For Code Coverage

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- **Fuzz and Produce Crashes**
- Identify Relevant Crashes

```
1   void process_string(const char *input_string) {
2       if (strlen(input_string) > 3) {
3           if (strlen(input_string == 6)) {
4               if (input_string[0] == 's') {
5                   if (strstr(input_string, "secret") != NULL) {
6                       int *ptr = NULL;
7                       *ptr = 1; // CRASH
8                   }
9               }
10          }
11      }
12  }
```

THE ATTACK CYCLE

# What is Code Coverage

Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

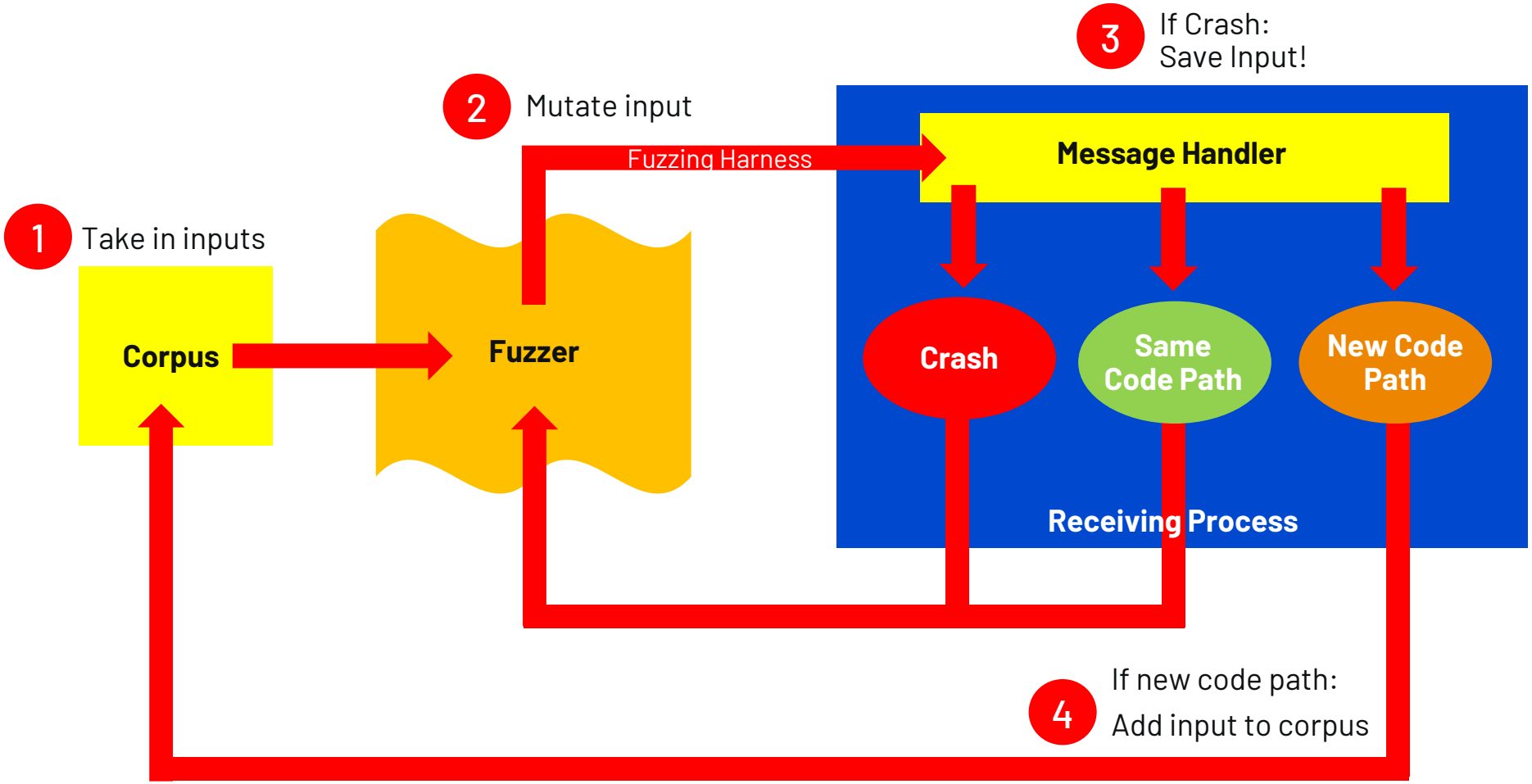
Identify Relevant Crashes

**Code coverage** traces a program's execution flow to identify new code paths.



# How Do We Determine Code Coverage?





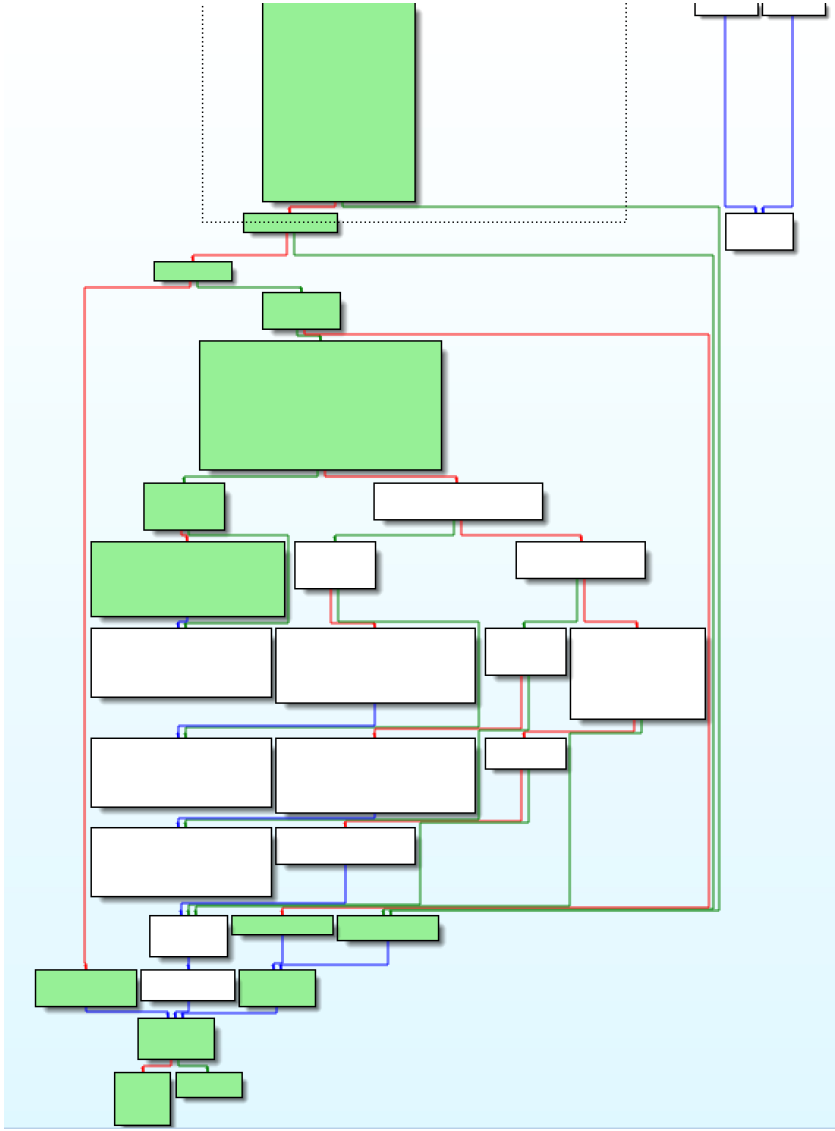


THE ATTACK CYCLE

# How Do We Determine Code Coverage?

Use instrumentation to monitor basic block execution

- Simple with source code:
  - AFL++ (https://github.com/AFLplusplus/AFLplusplus)
  - LibFuzzer (https://llvm.org/docs/LibFuzzer.html)
  - gCov (https://gcc.gnu.org/onlinedocs/gcc/Gcov.html)
- More difficult with black box binaries:
  - Frida (https://frida.re/)
  - TinyInst (https://github.com/googleprojectzero/TinyInst)
- Interpreting code coverage:
  - LightHouse for IdaPro/BinaryNinja (https://github.com/gaasedelen/lighthouse)



Identify an attack vector

Generate a Corpus of Inputs

Create a Fuzzing Harness

Fuzz and Produce Crashes

Identify Relevant Crashes

THE ATTACK CYCLE

# Actually Fuzzing!

My fuzzing setup

- Jackalope Fuzzer (https://github.com/googleprojectzero/Jackalope)
- Enable Apple's GuardMalloc
  - Restricted pages placed surrounding all allocations
  - `DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib`
- TinyInst for dynamic instrumentation to dump coverage
- LightHouse to interpret code coverage

THE ATTACK CYCLE

# Actually Fuzzing!

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- **Fuzz and Produce Crashes**
- Identify Relevant Crashes

```
fuzzychicken@Fuzzys-Mac Release % ./fuzzer -in ../../../modified_msg_ids
 -out audio-startup-modified-ids -t 200 -t1 5000 -delivery file -instrum
ent_module CoreAudio -target_module Harness -target_method _fuzz -nargs
1 -iterations 1000 -persist -loop -cmp_coverage -generate_unwind -dump_c
overage -target_env DYLD_INSERT_LIBRARIES=/usr/lib/libgmalloc.dylib  --
 ./Harness -f @@ -l /System/Library/Frameworks/CoreAudio.framework/Versi
ons/A/CoreAudio -s _HALB_MIGServer_server
Fuzzer version 1.00
63 input files read
Running input sample ../../../modified_msg_ids/1010000
GuardMalloc[Harness-3598]: Allocations will be placed on 16 byte boundar
ies.
GuardMalloc[Harness-3598]:  - Some buffer overruns may not be noticed.
GuardMalloc[Harness-3598]:  - Applications using vector instructions (e.
g., SSE) should work.
GuardMalloc[Harness-3598]: version 064555.99.1
Instrumented module CoreAudio, code size: 7462910

Total execs: 2
Unique samples: 0 (0 discarded)
Crashes: 0 (0 unique)
Hangs: 0
Offsets: 0
Execs/s: 2
GuardMalloc[Harness-3599]: Allocations will be placed on 16 byte boundar
ies.
GuardMalloc[Harness-3599]:  - Some buffer overruns may not be noticed.
GuardMalloc[Harness-3599]:  - Applications using vector instructions (e.
g., SSE) should work.
GuardMalloc[Harness-3599]: version 064555.99.1
Instrumented module CoreAudio, code size: 7462910
Exception at address 0x7ff85d79c63b
Access address: 0x108d80000
Exception in instrumented module CoreAudio 0x7ff81bbcd000
Code before:
47 ff ff c6 05 7e 72 bd 01 01
Code after:
41 89 5c 24 20 48 8b 05 b1 29 11 fe 48 8b 00 49
GuardMalloc[Harness-3600]: Allocations will be placed on 16 byte boundar
ies.
```

GuardMalloc

C++ Exception

Instrumentation

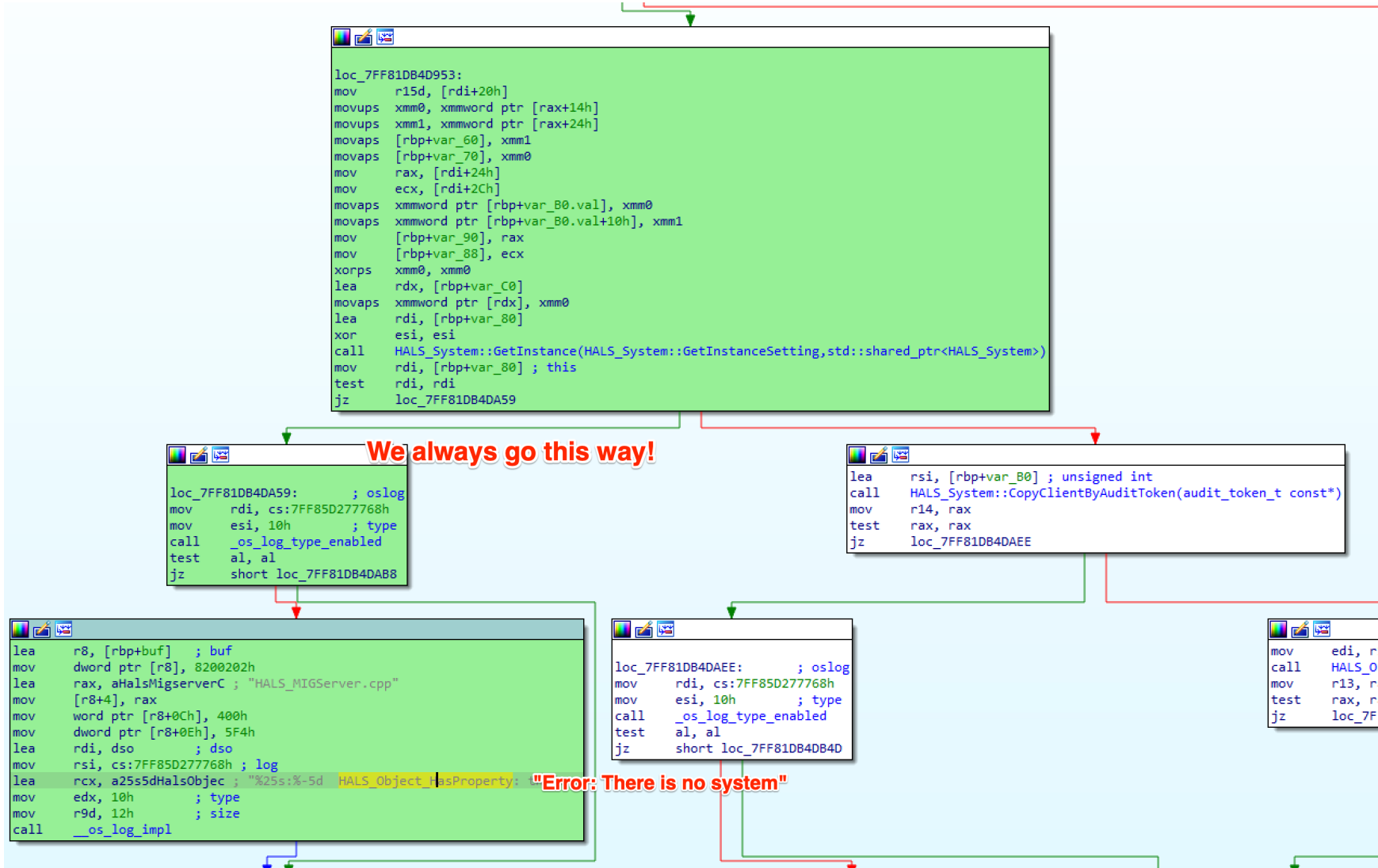
THE ATTACK CYCLE

# Regularly Check Code Coverage

- Identify an attack vector
- Generate a Corpus of Inputs
- Create a Fuzzing Harness
- Fuzz and Produce Crashes
- Identify Relevant Crashes

We can learn a lot from the code paths our fuzzer does and doesn't take

**Goal:** Cover as much of the binary as possible!





# Exploitable Versus Non-Exploitable Crashes



Exploitable:

- Crash on write
- Crash on execution
- Illegal instruction
- Heap corruption abort
- Stack trace contains **free**, **malloc**, etc.

Likely Non-Exploitable:

- Crash on read (could be used to leak memory, though)
- Handled exception
- Null pointer dereferences
- Stack recursion



# Exploitable Versus Non-Exploitable Crashes



## Exploitable:

- Crash on write
- Crash on execution
- Illegal instruction
- Heap corruption abort
- Stack trace contains **free**, **malloc**, etc.

## Likely Non-Exploitable:

- Crash on read (could be used to leak memory, though)
- Handled exception
- Null pointer dereferences
- Stack recursion

## Useful Tools:

- Apple's CrashWrangler (https://developer.apple.com/library/archive/technotes/tn2334/_index.html)
- CrashMon (https://github.com/ant4g0nist/crashmon)



# Exploitable Versus Non-Exploitable Crashes



## Exploitable:

- Crash on write
- Crash on execution
- Illegal instruction
- Heap corruption abort
- Stack trace contains **free**, **malloc**, etc.

## Likely Non-Exploitable:

- Crash on read (could be used to leak memory, though)
- Handled exception
- Null pointer dereferences
- Stack recursion

## Useful Tools:

- Apple's CrashWrangler (https://developer.apple.com/library/archive/technotes/tn2334/_index.html)
- CrashMon (https://github.com/ant4g0nist/crashmon)

## Crash Reproducibility

- Should be able to run input through harness and reproduce the crash

FUZZING TAKEAWAYS

# What We've Covered

- A crash course on fuzzing and Mach IPC mechanisms
- A walkthrough of the attack process:
  - Identifying an attack vector
  - Generating a corpus of fuzzing inputs
  - Writing a custom fuzzing harness
  - Fuzzing and producing crashes
  - Crash triaging
- Common pitfalls and things to consider
- Inspired you to do vulnerability research!

FUZZING TAKEAWAYS

# Next Steps

- Increase code coverage of Mach IPC handlers
  - Stateful Mach message fuzzing (determining message order when it matters)
  - Automatic initialization of Mach service binaries
- Open-source my Mach message dumper and fuzzing harness
  - Currently in progress, getting approval to release
- Collaborate with YOU!
  - Always looking for others to collaborate on research with

**Twitter:** @dillon_franke

**Blog:** https://dillonfrankesecurity.com

FUZZING TAKEAWAYS

## Questions

**Twitter:** @dillon_franke

**Blog:** https://dillonfrankesecurity.com

# Thank You!